

**МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ  
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»**

**КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ**

**МЕТОДИЧНІ ВКАЗІВКИ  
ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ  
З ДИСЦИПЛІНИ «ДИСКРЕТНА МАТЕМАТИКА»  
(для студентів спеціальності "Комп'ютерна інженерія")**

Ухвалено  
на засіданні кафедри ОТ  
протокол № 15 від 29.05.2024 р.

Погоджено  
на засіданні методичної  
комісії ФІОТ  
протокол № 10 від 21.06.2023 р.

Методичні вказівки до виконання лабораторних робіт з дисципліни «Дискретна математика» (для студентів спеціальності "Комп'ютерна інженерія"). /Укл.: Новотарський М. А.: НТУУ «КПІ» – 2024, 136 с.

Методичні вказівки до виконання лабораторних робіт з дисципліни «Дискретна математика» містять завдання до лабораторних робіт. До кожної роботи наведено методичні рекомендації з теорії, яка використовується при виконанні лабораторних робіт, приведені деякі алгоритми вирішення завдань, вимоги до виконання роботи, індивідуальні завдання та контрольні запитання.

Укладач: проф. Новотарський М.А.

Рецензент: проф. Стіренко С.Г.

## Лабораторна робота №1

**Тема:** «Множини: основні властивості та операції над ними, діаграми Венна».

**Мета:** вивчити основні аксіоми, закони і теореми теорії множин, навчитися застосовувати їх на практиці. Обчислити логічний вираз шляхом послідовного застосування операцій над множинами.

### Загальне завдання:

1. Повторити матеріал: «Бібліотека `tkinter` (віджети)» та виконати лабораторну роботу з застосуванням графічного інтерфейсу.
2. Спростити логічний вираз з застосуванням тотожностей алгебри множин.
3. В окремому модулі написати функцію обчислення початкового логічного виразу (1), вибраного відповідно до індивідуального варіанта.
4. В окремому модулі написати функцію обчислення спрощеного логічного виразу.
5. В окремому модулі написати функцію виконання логічної операції (2), вибраної відповідно до індивідуального варіанта.
6. В окремому модулі виконати порівняння результатів:
  - А) обчислення початкового та спрощеного виразу
  - Б) виконання логічної операції Вашою функцією та відповідною стандартною логічною операцією або функцією Python.

### Теоретичні основи:

#### 1.1. Властивості множин

**Множина** – є сукупність визначених об'єктів, різних між собою, об'єднаних за певною ознакою чи властивістю.

Множини позначають **великими** латинськими буквами. Об'єкти, що складають множини, називають елементами і позначають **малими** буквами латинського алфавіту.

На практиці часто застосовують такі позначення:

$N$  – множина натуральних чисел;

$P$  – множина додатних цілих чисел;

$Z$  – множина додатних і від'ємних цілих чисел, включаючи нуль;

$Q$  – множина раціональних чисел;

$R$  – множина дійсних чисел.

Якщо множина не містить жодного елемента, її називають порожньою і позначають  $\emptyset$ .

**Скінченна множина** – це така множина, кількість елементів якої може бути виражена скінченним числом, причому не важливо, чи можемо ми порахувати це число в даний момент.

**Нескінченна множина** – це така множина, що не є скінченною.

### Способи задавання множин:

- **перерахуванням**, тобто списком всіх елементів. Такий спосіб задавання прийнятний тільки при задаванні скінченних множин. Позначення списку – у фігурних дужках. Наприклад, множина, що з перших п'яти простих чисел  $A = \{2, 3, 5, 7, 11\}$ . Множина спортсменів університетської хокейної команди:  $B = \{\text{Іванов, Петров, Сидоров, Бубликов, Сирожкін, Волосянко}\}$ ;

- **процедурою**, що породжує і описує спосіб одержання елементів множини із уже отриманих елементів або з інших об'єктів. Наприклад, множина усіх цілих чисел, що є степенями двійки  $M_{2^n}, n \in N$ , де  $N$  - множина натуральних чисел, може бути представлена породжуючою процедурою, заданою двома правилами, названими рекурсивними: а)  $1 \in M_{2^n}$ ; б) якщо  $t \in M_{2^n}$ , тоді  $2t \in M_{2^n}$ ;

- **описом характеристичних властивостей**, які повинні мати елементи множини. Так, множину  $A$ , що складається з таких елементів  $x$ , які мають властивість  $P(x)$ , позначимо в такий спосіб:

$$A = \{x | P(x)\}.$$

Так, розглянута вище множина всіх цілих чисел, що є степенями числа 2, може бути записана як  $A = \{x | x = 2^n, n \in N\}$ . До  $A$  ще потрібно додати 1.

Якщо елемент  $a$  належить множині  $A$ , то пишуть  $a \in A$ . Якщо  $a$  не є елементом множини  $A$ , то пишуть  $a \notin A$ . Наприклад,  $5 \in \{1, 3, 5, 7\}$ , але  $4 \notin \{1, 3, 5, 7\}$ . Якщо  $A = \{x | \text{студентки групи ММ}_{21}\}$ , то  $\text{Іванова} \in A$ , а  $\text{Петров} \notin A$ .

**Підмножина.** Множину  $A$  називають підмножиною (або *включенням*) множини  $B$  ( $A \subseteq B$ ), якщо кожен елемент множини  $A$  є елементом множини  $B$ , тобто, якщо  $x \in A$ , то  $x \in B$ . Якщо  $A \subseteq B$  й  $A \neq B$ , то  $A$  називають строгою підмножиною й позначають  $A \subset B$ .

**Рівність множин.** Дві множини рівні ( $A = B$ ), якщо всі їхні елементи збігаються. Множини  $A$  і  $B$  рівні, якщо  $A \subseteq B$  і  $B \subseteq A$ .

**Потужність множини.** Кількість елементів у скінченній множині  $A$  називають *потужністю* множини  $A$  і позначають  $|A|$ .

**Універсальна множина**  $U$  є множина, що має таку властивість, що всі розглянуті множини є її підмножинами.

Варто розрізняти поняття **належності** елементів множини і **включення**! Так, наприклад, якщо множина  $A = \{1, 3, 6, 13\}$ , то  $3 \in A$ ,  $6 \in A$ , але  $\{3, 6\} \notin A$ , у той час як  $\{3, 6\} \subseteq A$ .

**Булеан.** Множину всіх підмножин, що складаються з елементів множини  $A$ , називають *булеаном*  $P(A)$ .

*Приклад булеану.* Нехай  $A = \{a, b, c, d\}$ . Визначити булеан множини  $A$ . Яка потужність множини  $P(A)$ ?

*Розв'язок:*

$$P(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}, \{a, b, c, d\}\}.$$

Потужність  $|P(A)| = 16$ .

## 1.2. Операції над множинами

**Об'єднання.** Об'єднанням множин  $A$  і  $B$  називають множину, що складається із всіх тих елементів, які належать хоча б одній з множин  $A$  або  $B$ . Об'єднання множин  $A$  і  $B$  позначають  $A \cup B$ . Це визначення рівносильне наступному:  $A \cup B = \{x | x \in A \text{ або } x \in B\}$ .

**Приклад.** Нехай  $A = \{2, 3, 5, 6, 7\}$ ,  $B = \{1, 2, 3, 7, 9\}$ . Знайти  $A \cup B$ .

Розв'язок:  $A \cup B = \{1, 2, 3, 5, 6, 7, 9\}$ .

**Перетин.** Перетином множин  $A$  і  $B$  називають множину, що складається із всіх тих елементів, які належать як множині  $A$ , так і множині  $B$ . Перетин множин  $A$  і  $B$  позначають  $A \cap B$ . Це визначення рівносильне наступному:  $A \cap B = \{x | x \in A \text{ і } x \in B\}$

**Приклад.** Нехай  $A = \{2, 3, 5, 6, 7\}$ ,  $B = \{1, 2, 3, 7, 9\}$ . Знайти  $A \cap B$ .

Розв'язок:  $A \cap B = \{2, 3, 7\}$ .

**Доповнення.** Доповненням (або абсолютним доповненням) множини  $A$  називають множину, що складається із всіх елементів універсальної множини, які не належать  $A$ . Доповнення множини  $A$  позначають  $\bar{A}$ . Це визначення рівносильне наступному:  $\bar{A} = U - A = \{x | x \in U \text{ и } x \notin A\}$ .

**Різниця.** Різницею множин  $A$  й  $B$  (або відносним доповненням) називають множину, що складається із всіх елементів множини  $A$ , які не належать  $B$ . Різницю множин  $A$  і  $B$  позначають  $A - B$  або  $A \setminus B$ . Це визначення рівносильне наступному:  $A - B = \{x | x \in A \text{ и } x \notin B\}$ .

**Приклад.** Нехай  $A = \{2, 3, 5, 6, 7\}$ ,  $B = \{1, 2, 3, 7, 9\}$ . Знайти  $A - B$ .

Розв'язок:  $A - B = \{5, 6\}$ .

**Симетрична різниця.** Симетричною різницею множин  $A$  і  $B$  називають множину, що складається з об'єднання всіх елементів, що належать множині  $A$  і не містяться в  $B$ , і елементів, що належать множині  $B$  і не містяться в  $A$ . Симетричну різницю множин  $A$  і  $B$  позначають  $A + B$  або  $A \Delta B$ . Це визначення рівносильне наступному:  $A \Delta B = (A \setminus B) \cup (B \setminus A)$ .

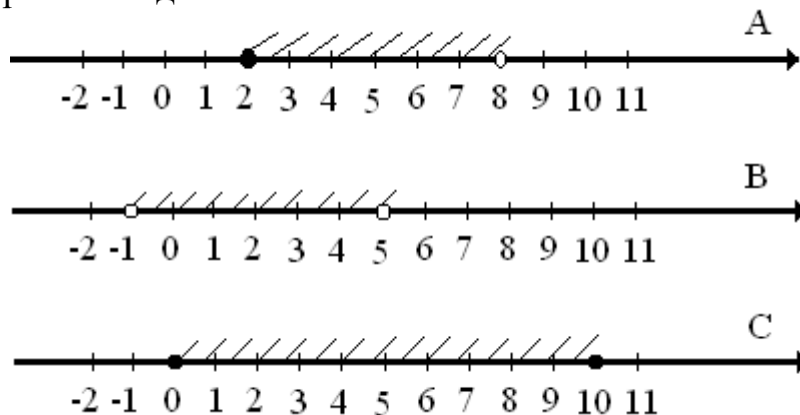
**Види операцій.** Операції, які виконують над однією множиною, називають *унарними*. Операції, які виконують над двома множинами, називають *бінарними*. Прикладом унарної операції є знаходження доповнення. Прикладами бінарних операцій є об'єднання, перетин, різниця, симетрична різниця.

**Приклад.** Нехай  $A = \{2, 3, 5, 6, 7\}$ ,  $B = \{1, 2, 3, 7, 9\}$ . Знайти  $A + B$ .

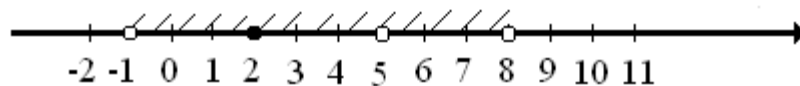
Розв'язок:  $A \Delta B = \{1, 5, 6, 9\}$ .

**Приклад.** Нехай  $A = [2, 8)$ ,  $B = (-1, 5)$ ;  $C = [0, 10]$ . Знайти  $A \cup B$ ,  $B \cap C$ ,  $C - A$ ,  $B \Delta C$ ,  $\overline{B}$ .

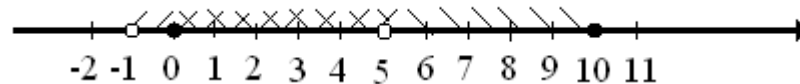
Розв'язок: Зобразимо задані множини на числовій осі



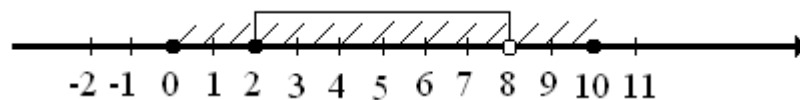
Тоді шукані множини будуть мати вигляд (рис. 1.1):



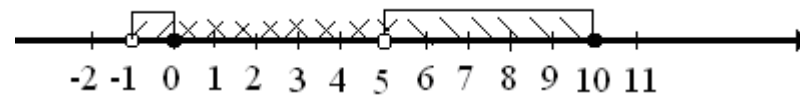
$$A \cup B = (-1, 8);$$



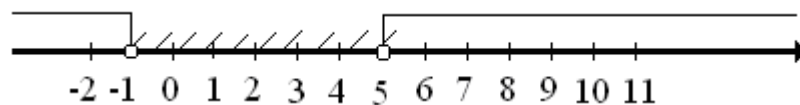
$$B \cap C = [0, 5);$$



$$C - A = [0, 2) \cup [8, 10];$$



$$B \Delta C = (-1, 0) \cup [5, 10];$$



$$\overline{B} = (-\infty, -1] \cup [5, \infty).$$

Рис. 1.1.

### 1.3. Діаграми Венна

Для графічної ілюстрації операцій над множинами даної універсальної множини  $U$  використовують діаграми Венна. Діаграма Венна – це зображення множини у вигляді геометричної множини, наприклад, кола. При цьому універсальну множину зображують у вигляді прямокутника. На рис. 1.2 зображені діаграми Венна для розглянутих операцій над множинами.

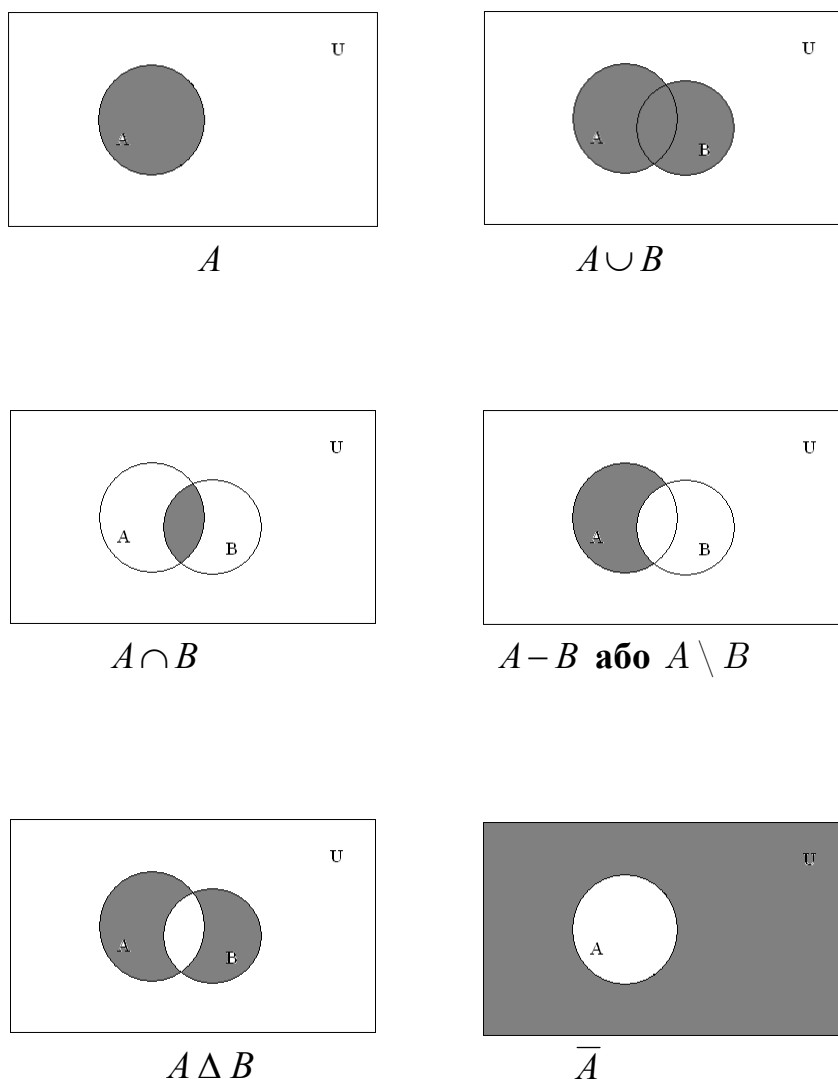


Рис. 1.2.

Для будь-яких підмножин  $A, B, C$  універсальної множини  $U$  справедливо наступне:

## 1.4. Тотожності алгебри множин

1. Комутативність об'єднання $A \cup B = B \cup A$	1. Комутативність перетину $A \cap B = B \cap A$
2. Асоціативність об'єднання $A \cup (B \cup C) = (A \cup B) \cup C$	2. Асоціативність перетину $A \cap (B \cap C) = (A \cap B) \cap C$
3. Дистрибутивність об'єднання відносно перетину $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	3. Дистрибутивність перетину відносно об'єднання $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
4. Закони дій з пустою та універсальною множинами  $A \cup \emptyset = A$  $A \cup \bar{A} = U$  $A \cup U = U$	4. Закони дій з пустою та універсальною множинами  $A \cap U = A$  $A \cap \bar{A} = \emptyset$  $A \cap \emptyset = \emptyset$
5. Закон ідемпотентності об'єднання Термін <b>ідемпотентність</b> означає властивість математичного об'єкта, яка проявляється в тому, що повторна дія над об'єктом <u>не змінює</u> його  $A \cup A = A$	5. Закон ідемпотентності перетину  $A \cap A = A$
6. Закон де Моргана  $\overline{A \cup B} = \bar{A} \cap \bar{B}$	6. Закон де Моргана  $\overline{A \cap B} = \bar{A} \cup \bar{B}$
7. Закон поглинання  $A \cup (A \cap B) = A$	7. Закон поглинання  $A \cap (A \cup B) = A$
8. Закон склеювання $(A \cap B) \cup (A \cap \bar{B}) = A$	8. Закон склеювання $(A \cup B) \cap (A \cup \bar{B}) = A$
9. Закон Порєцького $A \cup (\bar{A} \cap B) = A \cup B$	9. Закон Порєцького $A \cap (\bar{A} \cup B) = A \cap B$
10. Закон подвійного доповнення $\overline{\bar{A}} = A$	
11. Визначення операції «різниця»: $A \setminus B = A \cap \bar{B}$ .	
12. Визначення операції «симетрична різниця»: $A \Delta B = (A \setminus B) \cup (B \setminus A)$ .	

## 1.5. Вимоги до програмного забезпечення:

1. Лабораторна робота виконується з використанням скриптової мови програмування Python.
2. Для написання коду застосувати IDE PyCharm 3 Edu.
3. Для написання GUI застосувати бібліотеку tkinter.
4. Необхідно забезпечити ввід даних з клавіатури та з файлу.



## Зміст звіту:

### 1. Титульний лист повинен мати такий вигляд:

Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»  
Факультет інформатики та обчислювальної техніки  
Кафедра обчислювальної техніки

Дискретна математика  
Лабораторна робота №1  
«Множини: основні властивості та операції над ними, діаграми Венна».

Виконав:  
студент групи ІО-ХХ  
Прізвище, ім'я  
Залікова книжка № \_\_\_\_\_  
Перевірив Новотарський М.А.

Київ 2016р.

2. Мета лабораторної роботи та загальне завдання
3. Варіант виразу відповідно до індивідуального завдання
4. Короткі теоретичні відомості по темі, які відображають правила виконання логічних операцій, застосованих при виконанні лабораторної роботи.
5. Докладний опис спрощення логічного виразу з посиланням на тотожності алгебри множин, що застосовувалися при спрощенні.
6. Блок-схеми, які відповідають алгоритмам, що використані в лабораторній роботі.
7. Роздруківка того фрагменту тексту програми, який написаний індивідуально.
8. Роздруківка результатів виконання програми з контрольним прикладом
9. Аналіз результатів та висновки.

### 1.6. Контрольні запитання

1. Чим відрізняється множина від довільної сукупності елементів?
2. Назвіть способи задавання множин.
3. Які ви знаєте операції над множинами.
4. Чи може існувати множина, яка не містить жодного елемента.
5. Чи мають множини  $A$  і  $B$  однакові потужності, якщо  $A = \{1, 9, 25\}$ ,  $B = \{2300, 25, 1\}$ ?
6. Зобразіть діаграми Венна для вказаних викладачем операцій над двома множинами.
7. Запишіть вказане викладачем тотожне перетворення алгебри множин.
8. Дано дві множини  $A = \{1, 54, 12, 45, 11, 34\}$  і  $B = \{2, 11, 12, 13, 45, 54, 34\}$  та результат операції над ними:  $C = \{1, 2, 13\}$ . Вкажіть цю операцію.

### 1.7. Блок-схеми алгоритмів виконання операцій над множинами

#### Доповнення множини

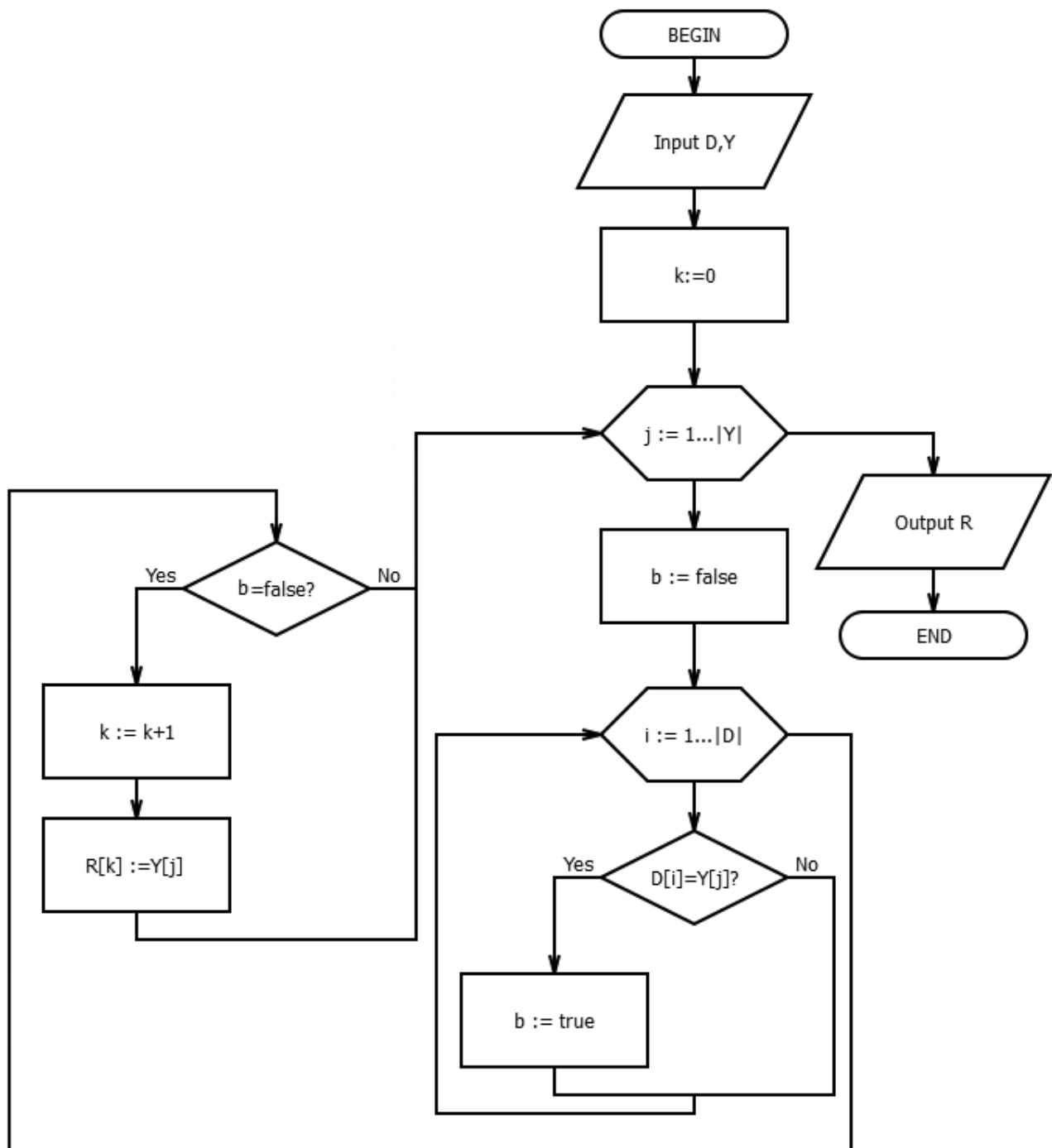


Рис.1.3 Блок-схема операції доповнення множини до універсальної

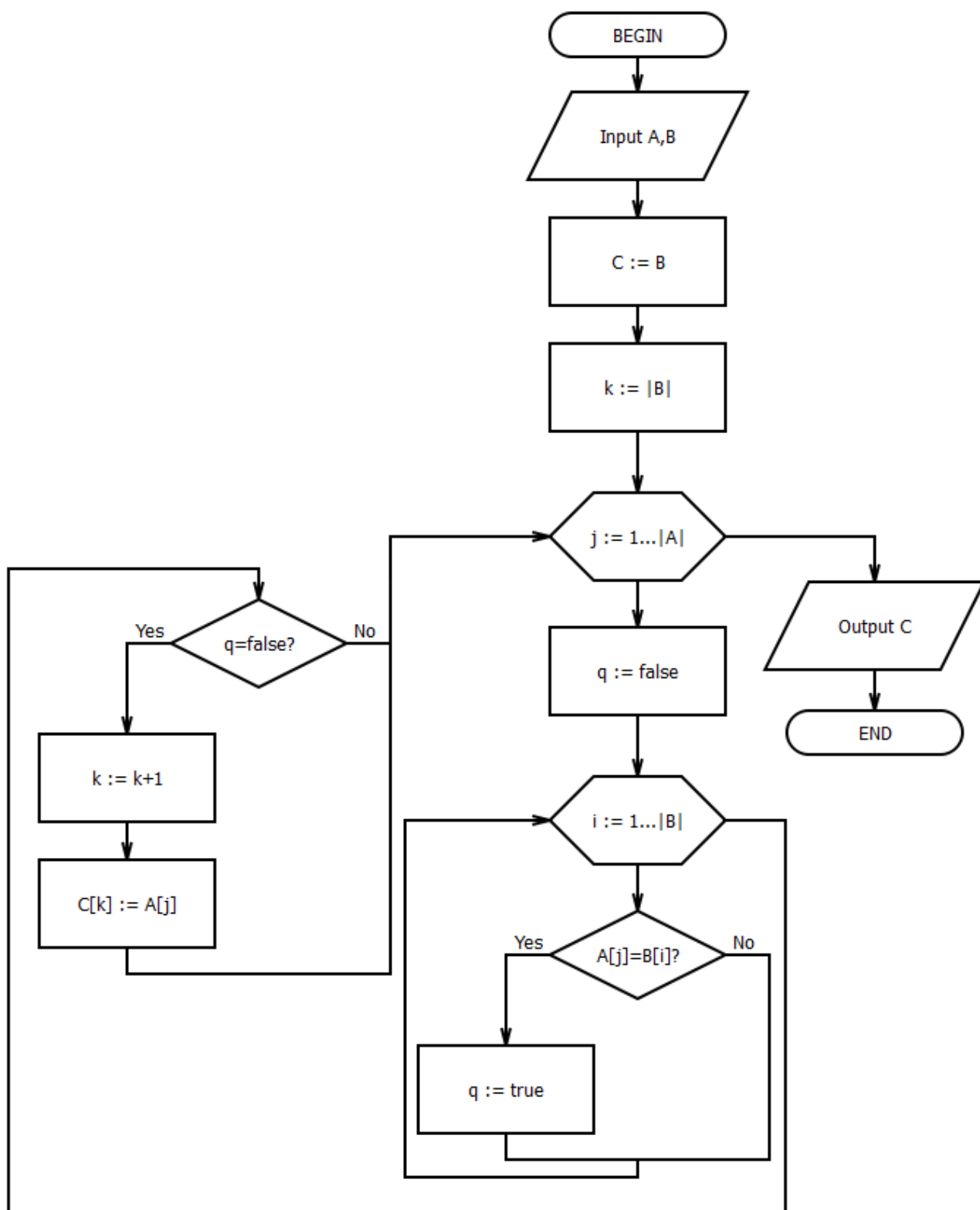


Рис.1.4. Блок-схема операції об'єднання двох множин

## Перетин двох множин

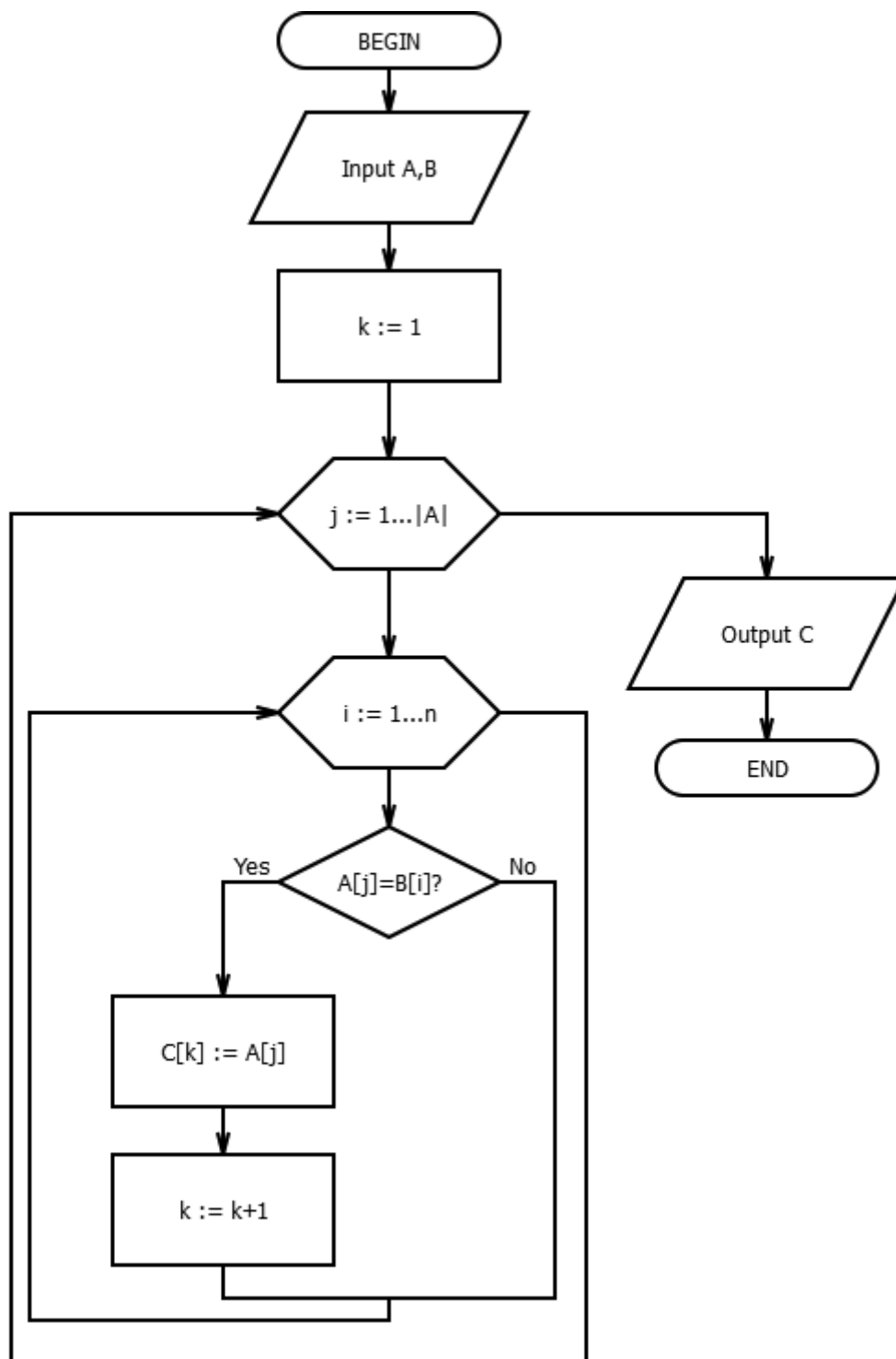


Рис. 1.5. Блок-схема операції перетину двох множин

### Різниця двох множин

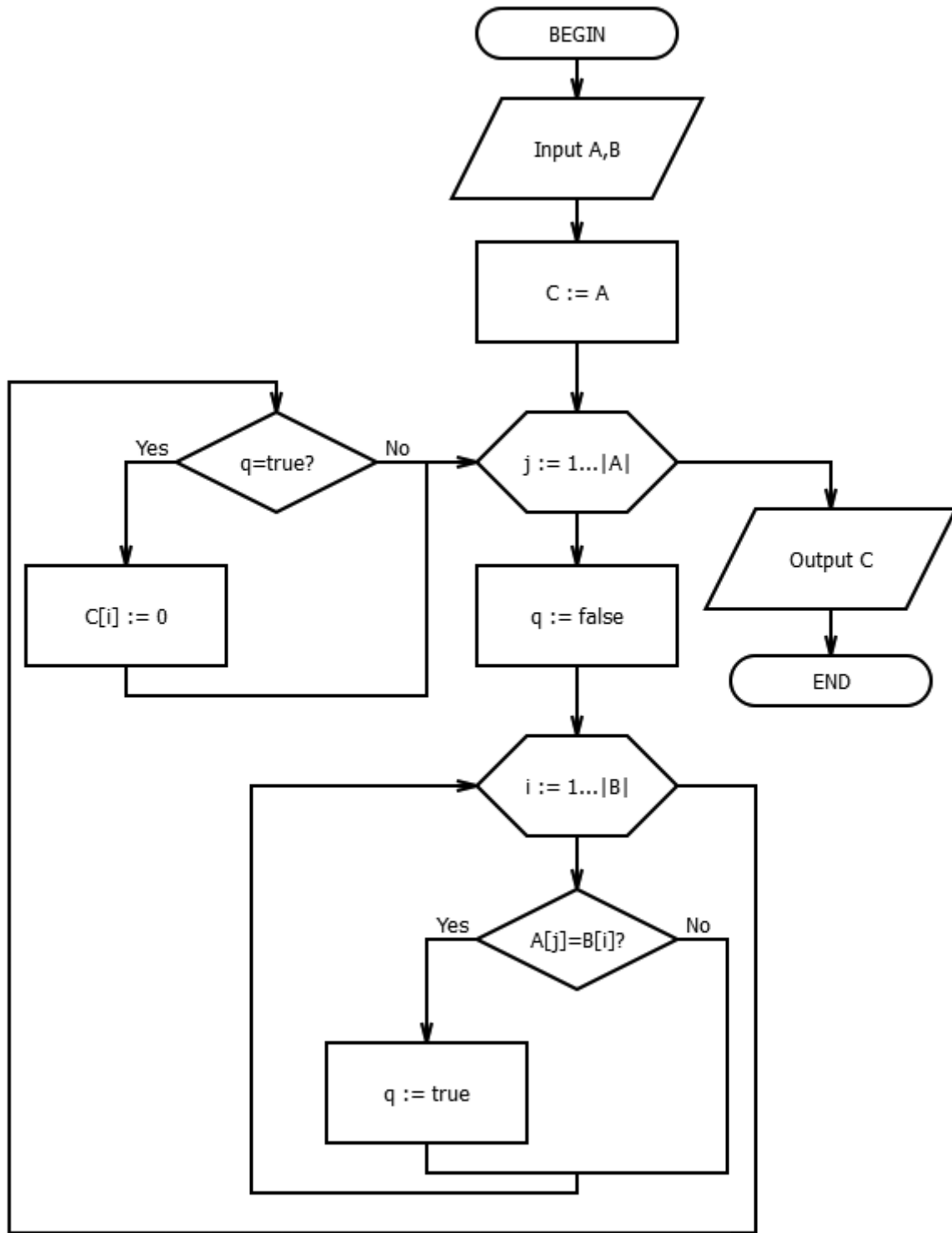


Рис. 1.6. Блок-схема операції різниці множин

## Симетрична різниця двох множин

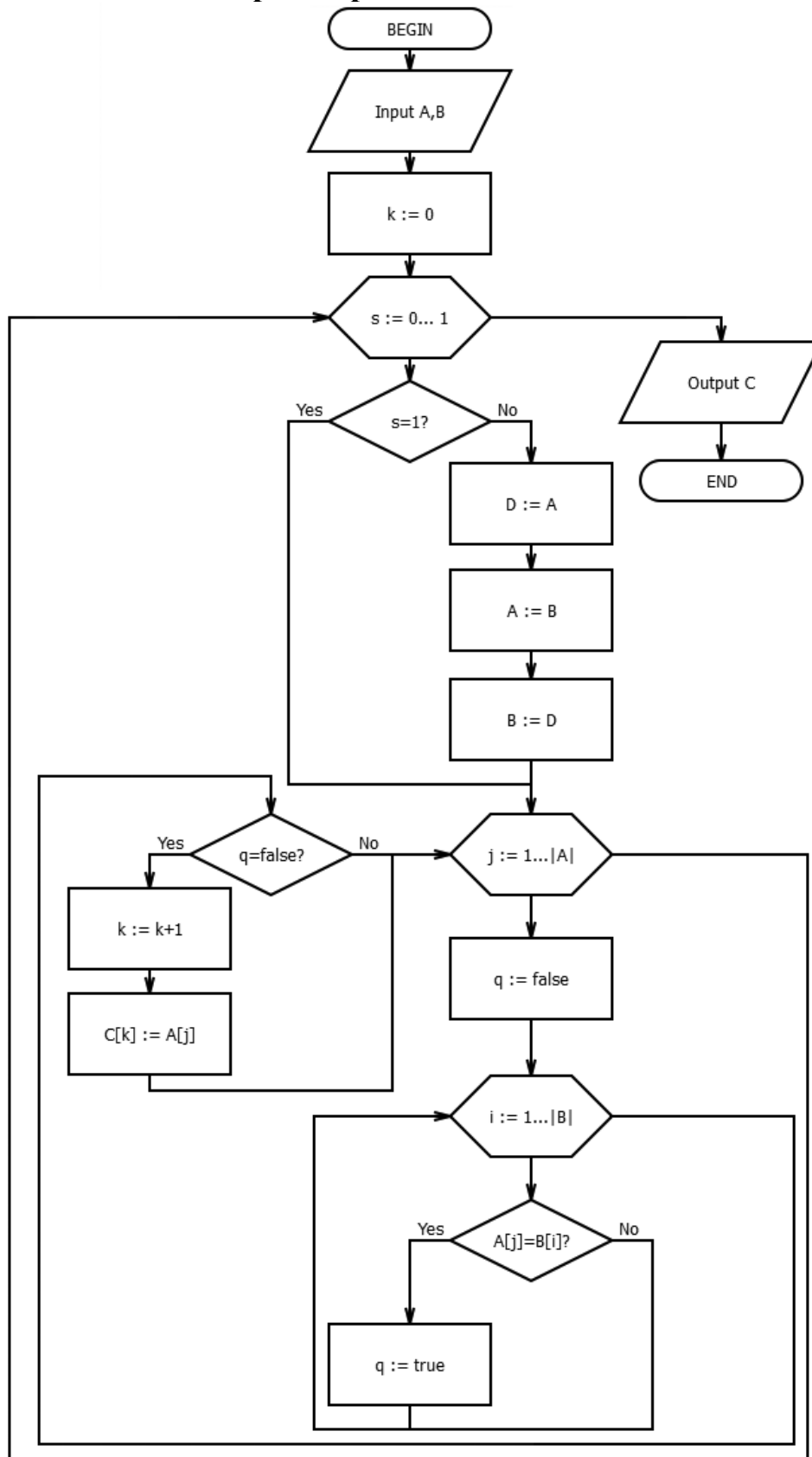


Рис. 1.7. Блок-схема операції симетричної різниці множин

## 1.8. Загальний порядок виконання лабораторної роботи

А) Вибрати номер  $Z$  індивідуального варіанта відповідно до виразу:

$Z = (i + G \% 60) \% 30 + 1$ , де  $i$  – номер у списку групи,  $G$  – числова складова назви групи.

Програма обчислення варіанта:

```
G=64
N=1
print("Моя група: IO -", G)
print("Мій номер у групі:", N)
print("Мій варіант:", (N+G%60)%30+1)
```

Б) Максимально спростити логічний вираз. Для спрощення використати тотожності алгебри множин та визначення логічних операцій. Спрощення є максимальним, якщо формула містить тільки одне входження кожної множини.

В) Створити блок-схему послідовності обчислення початкового логічного виразу.

Г) Створити блок-схему послідовності обчислення спрощеного логічного виразу.

Д) З використанням блок-схем створити проект, який містить модуль з функцією для обчислення початкового виразу (1) та модуль з функцією для обчислення спрощеного виразу.

Е) З використанням блок-схем, заданих у лабораторній роботі, створити модуль з функцією для виконання логічної операції (2), вибраної відповідно до варіанта.

Ж) В основному файлі виконати перевірку правильності спрощення виразу з виводом відповідного повідомлення.

З) Як елементи множин можуть бути використані числа від 0 до 255.

І) Лабораторну роботу виконувати з застосуванням мови Python та бібліотеки tkinter.

## 1.9. Вимоги до інтерфейсу

А) Програма повинна складатися з 5-ти вікон.

### Вміст вікна №1

1. Головне меню, яке повинно включати виклик вікна №2, вікна №3, вікна №4 та вікна №5.

2. Віджети виводу П.І.Б студента, номера групи, номера у списку та віджет виводу результатів обчислення варіанту відповідно до програми, що задана у завданні – пункт (А) загального порядку виконання лабораторної роботи).

3. Віджети для задавання потужності множин  $A$ ,  $B$  і  $C$ .

4. Віджети для формування випадковим чином множин  $A$ ,  $B$  і  $C$  заданої потужності.

5. Віджети, що дають можливість ручного вводу множин  $A$ ,  $B$  і  $C$ .

6. Віджет для задавання діапазону цілих чисел, які будемо вважати універсальною множиною.

### Вміст вікна №2

1. Віджети для відображення елементів множин  $A$ ,  $B$  і  $C$ .

2. Віджети запуску покрокового виконання початкового виразу (1).

Одним кроком вважати виконання однієї логічної операції.

3. Віджети відображення множин-операндів та множини-результату кожної логічної операції.

4. Віджет відображення множини D та віджет для виконання команди збереження даного результату у файлі.

### Вміст вікна №3

1. Віджети для відображення елементів множин A, B і C.

2. Віджети запуску покрокового виконання спрощеного логічного виразу.

Одним кроком вважати виконання однієї логічної операції.

3. Віджети відображення множин-операндів та множини-результату кожної логічної операції.

4. Віджет відображення множини D, та віджет для виконання команди збереження даного результату у файлі.

### Вміст вікна №4

1. Віджети для відображення елементів множин X і Y.

2. Віджет відображення результату Z виконання заданої у індивідуальному варіанті логічної операції (2) над множинами X і Y, яка виконана за допомогою написаної вами функції.

### Вміст вікна №5

1. Віджети для зчитування з файлу та відображення множини, яка є результатом обчислення початкового виразу.

2. Віджети для зчитування з файлу та відображення множини, яка є результатом обчислення спрощеного виразу.

3. Віджети для зчитування з файлу та відображення множини Z, яка є результатом обчислення операції над множинами X і Y, яка виконана за допомогою написаної вами функції.

4. Віджети для запуску та відображення логічної операції над множинами X і Y, яка є результатом обчислення Z з використанням стандартної функції Python.

5. Віджети для порівняння результатів виводу у пунктах 1 та 2 та у пунктах 3 та 4.

### Варіанти для виконання лабораторної роботи

При виконанні індивідуального завдання лабораторної роботи використати варіант з таблиці:

№ вар		Вираз для обчислення
1	(1)	$D = ((A \setminus B) \cup (B \cap A)) \setminus (C \cup B)$
	(2)	$X = A; Y = B; Z = X \setminus Y$
2	(1)	$D = A \cap (A \setminus (A \setminus B)) \cup C$
	(2)	$X = A; Y = B; Z = X \cap Y$



№ вар		Выраз для обчислення
3	(1)	$D = (A \cup (B \setminus A)) \setminus C$
	(2)	$X = A; Y = C; Z = X \cup Y$
4	(1)	$D = \overline{(A \cup B)} \cap (\overline{B} \cup \overline{C})$
	(2)	$X = \overline{A}; Y = C; Z = X \Delta Y$
5	(1)	$D = (A \cup (\overline{A} \cap B)) \Delta (C \cup (C \cap B))$
	(2)	$X = B; Y = C; Z = X \setminus Y$
6	(1)	$D = ((A \cap B) \cup (A \setminus B)) \Delta (C \cup B)$
	(2)	$X = \overline{C}; Y = A; Z = X \cup Y$
7	(1)	$D = C \cap (A \cap \overline{B}) \cap (C \cup B)$
	(2)	$X = \overline{A}; Y = \overline{B}; Z = X \cap Y$
8	(1)	$D = ((A \cap \overline{B}) \cup (B \cap \overline{A})) \cap (C \cup B) \cap C$
	(2)	$X = C; Y = A; Z = X \Delta Y$
9	(1)	$D = (((A \cup \overline{A}) \cap A) \setminus B \cup B) \cap \overline{(C \cup (C \cap B))}$
	(2)	$X = A; Y = C; Z = X \setminus Y$
10	(1)	$D = (A \cap B) \cup ((A \cap \overline{C}) \cup (\overline{A} \cap B))$
	(2)	$X = \overline{C}; Y = B; Z = X \cap Y$
11	(1)	$D = A \cap (A \cup \overline{B}) \cap (C \cup (\overline{C} \cap B))$
	(2)	$X = A; Y = \overline{C}; Z = X \cup Y$
12	(1)	$D = ((A \cap \overline{B}) \cup B \setminus A) \cup C$
	(2)	$X = \overline{B}; Y = A; Z = X \Delta Y$
13	(1)	$D = (\overline{A} \cap (A \cup \overline{B})) \setminus C$
	(2)	$X = C; Y = \overline{B}; Z = X \setminus Y$
14	(1)	$D = (A \cap (B \cup A)) \cap \overline{(C \cup B)} \cap \overline{C}$
	(2)	$X = \overline{B}; Y = \overline{A}; Z = X \cap Y$
15	(1)	$D = (A \cup (B \cap C)) \cap (\overline{A} \cup C)$
	(2)	$X = \overline{A}; Y = B; Z = X \cup Y$
16	(1)	$D = A \Delta (B \setminus ((C \cup A) \cap (C \cup \overline{A})))$
	(2)	$X = A; Y = \overline{C}; Z = X \Delta Y$

№ вар		Выраз для обчислення
17	(1)	$D = (A \cup B) \cap (\overline{A \cap B}) \cup (\overline{B \cup C})$
	(2)	$X = C; Y = \overline{B}; Z = X \setminus Y$
18	(1)	$D = \overline{A} \cup \overline{B} \cup (\overline{A \cap B}) \cup (\overline{B \cap C}) \cup \overline{C}$
	(2)	$X = \overline{A}; Y = \overline{B}; Z = X \cap Y$
19	(1)	$D = C \cup (\overline{A \cap B}) \cap (\overline{B \cap A}) \cap (\overline{A \cup B})$
	(2)	$X = C; Y = \overline{A}; Z = X \cup Y$
20	(1)	$D = ((A \cup B) \cup C \cup (B \cup C) \cup A)$
	(2)	$X = B; Y = \overline{C}; Z = X \Delta Y$
21	(1)	$D = A \cap (\overline{B \cup C}) \cup (\overline{A \cap C})$
	(2)	$X = C; Y = B; Z = X \setminus Y$
22	(1)	$D = B \Delta C \cup (B \cap C) \Delta (((A \setminus B) \cap B) \Delta A)$
	(2)	$X = \overline{B}; Y = A; Z = X \cap Y$
23	(1)	$D = ((A \cup B) \cup (A \cup \overline{B})) \cap \overline{B} \cap A \cap (\overline{A \cup C})$
	(2)	$X = B; Y = \overline{A}; Z = X \cup Y$
24	(1)	$D = \overline{A} \cup B \cup \overline{C} \cup (B \cap \overline{C}) \cup (\overline{A \cap C}) \cup (A \cap B)$
	(2)	$X = \overline{B}; Y = \overline{A}; Z = X \Delta Y$
25	(1)	$D = (A \cap B) \cup (A \cap \overline{B}) \cup (C \cup A) \cap (A \cup \overline{B})$
	(2)	$X = \overline{B}; Y = \overline{C}; Z = X \setminus Y$
26	(1)	$D = \overline{A} \cup (\overline{A \cup B}) \cap (\overline{A \cup C}) \cup \overline{B} \cap ((B \cap C) \cup (B \cap \overline{C}))$
	(2)	$X = \overline{C}; Y = A; Z = X \cap Y$
27	(1)	$D = (A \cap B) \cup (C \cap B) \cup (\overline{A \cap B}) \cup (\overline{B \cap C})$
	(2)	$X = A; Y = \overline{B}; Z = X \cup Y$
28	(1)	$D = (\overline{A \cup B}) \cap (\overline{A \cup C}) \cap (\overline{B \cup C})$
	(2)	$X = A; Y = \overline{C}; Z = X \Delta Y$
29	(1)	$D = ((A \cap \overline{B}) \cup (\overline{A \cap B})) \cap (\overline{C} \cap (\overline{C} \cup B))$
	(2)	$X = \overline{B}; Y = \overline{A}; Z = X \setminus Y$
30	(1)	$D = \overline{C} \cap (A \setminus C) \cap (B \setminus C) \cap (\overline{C} \cup B)$
	(2)	$X = C; Y = B; Z = X \cap Y$

## Лабораторна робота №2

**Тема:** «Бінарні відношення та їх основні властивості, операції над відношеннями».

**Мета:** вивчити основні властивості бінарних відношень та оволодіти операціями над бінарними відношеннями.

### Загальне завдання:

1. Написати в окремому модулі функцію для формування несуперечливих бінарних відношень.
2. Написати в окремому модулі функції виконання логічних операцій над бінарними відношеннями.
3. Пояснити правило формування несуперечливих відношень відповідно до Вашого варіанту.

### Теоретичні основи:

#### **2.1. Основні означення**

**Упорядкована пара предметів** – це сукупність, що складається із двох предметів, розташованих у деякому певному порядку. При цьому впорядкована пара має наступні властивості:

а) для будь-яких двох предметів  $x$  і  $y$  існує об'єкт, який можна позначити як  $\langle x, y \rangle$ , названий упорядкованою парою;

б) якщо  $\langle x, y \rangle$  і  $\langle u, v \rangle$  – упорядковані пари, то  $\langle x, y \rangle = \langle u, v \rangle$  тоді і тільки тоді, коли  $x = u$ ,  $y = v$ .

При цьому  $x$  будемо називати першою координатою, а  $y$  – другою координатою впорядкованої пари  $\langle x, y \rangle$ .

**Бінарним** (або **двомісним**) відношенням  $R$  називають підмножину впорядкованих пар, тобто множину, кожен елемент якої є впорядкованою парою.

Якщо  $R$  є деяким відношенням, це записують як  $\langle x, y \rangle \in R$  або  $xRy$ .

Один з типів відношень – це множина всіх таких пар  $\langle x, y \rangle$ , що  $x$  є елементом деякої фіксованої множини  $X$ , а  $y$  – елементом деякої фіксованої множини  $Y$ . Таке відношення називають *прямим* або *декартовим добутком*.

**Декартовим добутком**  $X \times Y$  множин  $X$  і  $Y$  є множина  $\{\langle x, y \rangle \mid x \in X, y \in Y\}$ .

При цьому множину  $X$  називають *областю визначення* відношення  $R$ , а  $Y$  – його *областю значень*:

$$D(R) = \{x \mid \langle x, y \rangle \in R\}; \quad E(R) = \{y \mid \langle x, y \rangle \in R\}$$

**Бінарним** відношенням  $R$  називають підмножину пар  $\langle x, y \rangle \in R$  прямого добутку  $X \times Y$ , тобто  $R \subseteq X \times Y$ .

У силу визначення бінарних відношень, як **спосіб їх задавання** можуть бути використані будь-які способи задавання множин. Відношення, визначені на скінченних множинах, зазвичай задають:

1. *Списком (перерахуванням)* упорядкованих пар, для яких це відношення виконується.

2. *Матрицею* – бінарному відношенню  $R \subseteq X \times X$ , де  $X = \{x_1; x_2; \dots; x_n\}$  відповідає квадратна матриця порядку  $n$ , кожен елемент  $a_{ij}$  якої дорівнює 1, якщо між  $x_i$  й  $x_j$  є відношення  $R$ , і 0 у протилежному випадку, тобто:

$$a_{ij} = \begin{cases} 1, & \text{якщо } x_i R x_j, \\ 0, & \text{у протилежному випадку.} \end{cases}$$

**Приклад.** Нехай  $A = \{1, 2, 3\}$ ,  $B = \{2, 3, 4\}$ . Знайти декартовий добуток множин  $A \times B$  й  $B \times A$ . Записати  $(A \times B) - (B \times A)$ ,  $(A \times B) \cap (B \times A)$ ,  $(A \times B) + (B \times A)$ .

Рішення:  $A \times B = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle, \langle 3, 4 \rangle\}$ ;

$B \times A = \{\langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle, \langle 4, 1 \rangle, \langle 4, 2 \rangle, \langle 4, 3 \rangle\}$ ;

$(A \times B) - (B \times A) = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 4 \rangle, \langle 3, 4 \rangle\}$ ;

$(A \times B) \cap (B \times A) = \{\langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle\}$ ;

$(A \times B) + (B \times A) = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 4 \rangle, \langle 3, 4 \rangle, \langle 2, 1 \rangle, \langle 3, 1 \rangle, \langle 4, 1 \rangle, \langle 4, 2 \rangle, \langle 4, 3 \rangle\}$ .

## 2.2. Властивості бінарних відношень

1. Відношення  $R$  на  $A \times A$  називають **рефлексивним**, якщо  $\langle a, a \rangle \in R$  для кожного  $a \in A$ . Головна діагональ матриці такого відношення містить тільки одиниці.

2. Відношення  $R$  на  $A \times A$  називають **антирефлексивним**, якщо для жодного  $a \in A$  не виконується  $\langle a, a \rangle \in R$ , тобто із  $\langle a, b \rangle \in R$  потрібно, щоб  $a \neq b$ . Головна діагональ матриці такого відношення містить тільки нулі.

3. Відношення  $R$  на  $A \times A$  називають **симетричним**, якщо для всіх  $a, b \in R$  з умови  $\langle a, b \rangle \in R$  потрібно, щоб  $\langle b, a \rangle \in R$ . Матриця симетричного відношення симетрична відносно головної діагоналі, тобто  $c_{ij} = c_{ji}$  для всіх  $i$  і  $j$ .

4. Відношення  $R$  на  $A \times A$  називають **антисиметричним**, якщо для всіх  $a, b \in R$ , з умов  $\langle a, b \rangle \in R$  і  $\langle b, a \rangle \in R$  потрібно, щоб  $a = b$ , тобто для жодних елементів  $a$  і  $b$ , що розрізняються ( $a \neq b$ ), не виконуються одночасно відношення  $\langle a, b \rangle \in R$  і  $\langle b, a \rangle \in R$ . У матриці антисиметричного відношення відсутні одиниці, симетричні відносно головної діагоналі.

5. Відношення  $R$  на  $A \times A$  називають **транзитивним**, якщо для будь-яких  $a, b, c$  з умов  $\langle a, b \rangle \in R$  і  $\langle b, c \rangle \in R$  випливає  $\langle a, c \rangle \in R$ . У матриці такого відношення повинна виконуватися наступна умова: якщо в  $i$ -тому рядку і в  $j$ -тому стовпці стоїть одиниця, тобто  $c_{ij} = 1$ , то всім одиницям в  $j$ -тому рядку і  $k$ -тому стовпці ( $c_{jk} = 1$ ) повинні відповідати одиниці в  $i$ -тому рядку і у тих же  $k$ -тих стовпцях, тобто  $c_{ik} = 1$  (і, можливо, в інших стовпцях).

6. Бінарне відношення називають **еквівалентним**, якщо воно рефлексивне, симетричне і транзитивне.

### 2.3. Операції над відношеннями

Оскільки відношення на множині  $A$  задають підмножинами  $R \subseteq A \times B$ , то для них визначні ті ж операції, що й над множинами, а саме:

1. **Об'єднання:**  $R_1 \cup R_2 = \{\langle a, b \rangle \mid \langle a, b \rangle \in R_1 \text{ або } \langle a, b \rangle \in R_2\}$ .

2. **Перетин:**  $R_1 \cap R_2 = \{\langle a, b \rangle \mid \langle a, b \rangle \in R_1 \text{ і } \langle a, b \rangle \in R_2\}$ .

3. **Різниця:**  $R_1 \setminus R_2 = \{\langle a, b \rangle \mid \langle a, b \rangle \in R_1 \text{ і } \langle a, b \rangle \notin R_2\}$ .

4. **Доповнення:**  $\overline{R} = U \setminus R$ , де  $U = A \times B$ .

Крім того, необхідно визначити інші операції над бінарними відношеннями.

5. **Обернене відношення**  $R^{-1}$ .

Якщо  $\langle a, b \rangle \in R$  – відношення, то відношення  $R^{-1}$  називають **оберненим відношенням** до даного відношення  $R$  тоді й тільки тоді, коли  $R^{-1} = \{\langle b, a \rangle \mid \langle a, b \rangle \in R\}$ .

Наприклад, якщо  $R$  – “бути старішим”, то  $R^{-1}$  – “бути молодшим”; якщо  $R$  – “бути дружиною”, то  $R^{-1}$  – “бути чоловіком”.

Нехай  $R = \{\langle a, b \rangle \mid b \text{ є родич } a\}$  або  $R = \{\langle x, y \rangle \mid x^2 + y^2 = 1\}$ . У такому випадку  $R = R^{-1}$ .

Нехай  $R \subseteq A \times B$  – відношення на  $A \times B$ , а  $S \subseteq B \times C$  – відношення на  $B \times C$ .

**Композицією** відношень  $R$  і  $S$  називають відношення  $T \subseteq A \times C$ , таке, що  $T = \{\langle a, c \rangle \mid \text{існує такий елемент } b \text{ з } B, \text{ що } \langle a, b \rangle \in R \text{ і } \langle b, c \rangle \in S\}$ .

Цю множину позначають  $T = S \circ R$ .

Зокрема, якщо відношення  $R$  визначене на множині  $A$  ( $R \subseteq A^2$ ), то композицію визначають як

$$R \circ R = R^{(2)} = \{\langle a, c \rangle \mid \langle a, b \rangle, \langle b, c \rangle \in R\}.$$

**Транзитивним замиканням**  $R^0$  називають множину, що складається з таких і тільки таких пар елементів  $a, b$  з  $A$ , тобто  $\langle a, b \rangle \in R^0$ , для яких в  $A$  існує ланцюжок з  $n + 2$  ( $n \geq 0$ ) елементів  $A: a, c_1, c_2, \dots, c_n, b$ , між сусідніми елементами якої виконується відношення  $R: \langle a, c_1 \rangle \in R, \langle c_1, c_2 \rangle \in R, \dots, \langle c_n, b \rangle \in R$ , тобто

$$R^0 = \{\langle a, b \rangle \mid \langle a, c_1 \rangle, \langle c_1, c_2 \rangle, \dots, \langle c_n, b \rangle \in R\}.$$

Наприклад, для відношення  $R$  – “бути дочкою” композиція  $R \circ R = R^{(2)}$  – “бути онукою”,  $R \circ R \circ R = R^{(3)}$  – “бути правнучкою” і т. ін. Тоді об'єднання всіх цих відношень є транзитивним замиканням  $R^0$  – “бути прямим нащадком”.

Якщо відношення  $R$  транзитивне, то  $R^0 = R$ .

**Транзитивне замикання**  $R^0$  на  $R$  є найменшим транзитивним відношенням на  $A$ , що містить  $R$  як підмножину.

Процедура обчислення транзитивного замикання  $R^0$  для відношення  $R \in A \times A$ :

1) присвоїти  $R_i \leftarrow R$ ;

2) обчислити  $R_1 \cup R_1^{(2)} = R_1 \cup R_1$ , присвоїти  $R_2 \leftarrow R_1^{(2)}$ ;

3) порівняти  $R_1$  і  $R_2$ . Якщо  $R_1 = R_2$ , то перейти до кроку 4, якщо  $R_1 \neq R_2$ , то присвоїти  $R_1 \leftarrow R_2$  і перейти до кроку 2;

4)  $R_1 = R_2 = R^0$ .

#### 8. Рефлексивне замикання:

Нехай тотожне відношення  $E = \{\langle a, a \rangle \mid a \in A\}$ . Тоді *рефлексивне замикання* визначають як  $R^* = R^0 \cup E$ .

Якщо  $R$  транзитивне і рефлексивне, то  $R^* = R$ .

Рефлексивне замикання  $R^*$  на  $R$  є найменшим рефлексивним відношенням на  $A$ , що містить  $R$  як підмножину.

### 2.4. Вимоги до програмного забезпечення:

1. Лабораторна робота виконується з використанням скриптової мови програмування Python.
2. Для написання коду застосувати IDE PyCharm 3 Edu.
3. Для написання GUI застосувати бібліотеку tkinter.
4. Необхідно забезпечити ввід даних з клавіатури та з файлу.

#### Зміст звіту:

1. Титульний лист повинен мати такий вигляд:

Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»  
Факультет інформатики та обчислювальної техніки  
Кафедра обчислювальної техніки

Дискретна математика  
Лабораторна робота №2  
«Бінарні відношення та їх основні властивості, операції над відношеннями».

Виконав:  
студент групи ІО-ХХ  
Прізвище, ім'я  
Залікова книжка № \_\_\_\_\_  
Перевірів Новотарський М.А.

Київ 2016р.

2. Мета лабораторної роботи та загальне завдання
3. Варіант виразу відповідно до індивідуального завдання
4. Короткі теоретичні відомості по темі, які відображають правила виконання логічних операцій, застосованих при виконанні лабораторної роботи.
5. Блок-схеми, які відповідають алгоритмам, що використані в лабораторній роботі.
6. Роздрукована копія того фрагменту тексту програми, який написаний індивідуально.
7. Роздрукована копія результатів виконання програми з контрольним прикладом
8. Аналіз результатів та висновки.

## Контрольні запитання

1. Дати визначення бінарного відношення.
2. Способи задавання відношень.
3. Властивості бінарних відношень.
4. Визначення композиції відношень.

## 2.5. Етапи виконання роботи.

*Етап 1.* Створити програму, яка коректно формує відношення у відповідності з варіантом завдання та виконує операції над цими відношеннями.

*Етап 2.* Ввести елементи множин  $A$  та  $B$ . Наприклад:  $A = \{\text{Антоніна, Оксана, Галина, Ольга, Світлана, Петро, Тетяна, Іван, Катерина, Олег}\}$ ,  $B = \{\text{Борис, Василь, Максим, Ольга, Тетяна, Іван, Аркадій, Артем, Оксана, Петро}\}$ .

*Етап 3.* Задати програмно відношення  $S$  і  $R$  між елементами множин  $A$  і  $B$ .

Наприклад, використовуючи варіант:

$aSb$ , якщо  $a$  сестра  $b$ .  $aRb$ , якщо  $a$  дружина  $b$

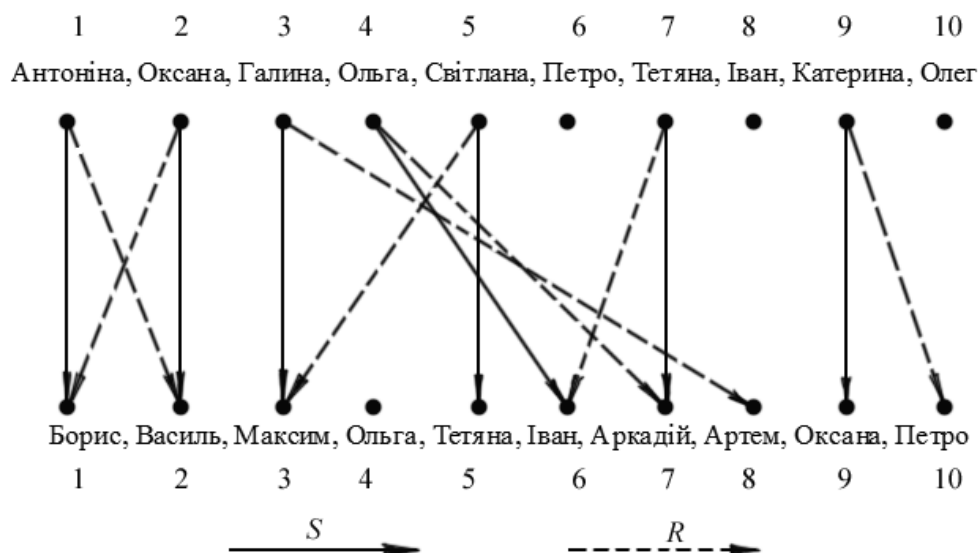


Рис. 2.1. Приклад відношення  $S$  і  $R$

*Етап 4.* Виконати програмно перевірку коректності задавання відношень.

Наприклад:

$S = \{\langle 1,1 \rangle, \langle 2,2 \rangle, \langle 3,3 \rangle, \langle 4,6 \rangle, \langle 5,5 \rangle, \langle 7,7 \rangle, \langle 9,9 \rangle\}$  – елементи відношення коректні,

$R = \{\langle 1,2 \rangle, \langle 2,1 \rangle, \langle 3,8 \rangle, \langle 4,7 \rangle, \langle 5,3 \rangle, \langle 7,6 \rangle, \langle 9,10 \rangle\}$  – елементи відношення коректні.

Приклади некоректних елементів відношень:

$\langle 1,1 \rangle \notin R$  – Антоніна не може бути дружиною Бориса, оскільки вона – його сестра.

$\langle 3,4 \rangle \notin R$  – Ольга – особа жіночої статі, тому не може бути дружиною Галини.

$\langle 6,9 \rangle \notin R$  – Петро – особа чоловічої статі, тому не може бути дружиною Оксани.

$\langle 4,4 \rangle \notin S$  – Ольга не може бути сестрою сама собі, і т. д.

## 2.6. Загальний порядок виконання лабораторної роботи

1. Вибрати номер  $Z$  індивідуального варіанта відповідно до виразу:

$Z=(i+G\%60)\%30+1$ , де  $i$  – номер у списку групи,  $G$  – числова складова назви групи.

Програма обчислення варіанта:

```
G=64
N=1
print("Моя група: IO -", G)
print("Мій номер у групі:", N)
print("Мій варіант:", (N+G%60)%30+1)
```

2. За індивідуальним варіантом побудувати алгоритм, який встановлює одночасно несуперечливі відношення  $S$  та  $R$  для двох довільних множин.

3. Відношення  $S$  та  $R$  представити у вигляді матриць або графів.

4. Результат операцій над відношеннями також представити у вигляді матриць або графів.

## 2.7. Вимоги до інтерфейсу

А) Програма повинна складатися з 4-х вікон.

### Вміст вікна №1

1. Головне меню, яке повинно включати виклик вікна №2, вікна №3 та вікна №4.

2. Віджети виводу П.І.Б студента, номера групи, номера у списку та віджет виводу результатів обчислення варіанту відповідно до програми, що задана у завданні – (пункт 1 загального порядку виконання лабораторної роботи).

### Вміст вікна №2

3. Віджет Listbox зі списком жіночих імен з можливістю вибору з нього довільної кількості імен з наступним копіюванням цих імен у віджет Listbox списку множини А або В, в залежності від стану віджета Radiobutton, який вказує у який список відбувається копіювання імені.

4. Віджет Listbox зі списком чоловічих імен з можливістю вибору з нього довільної кількості імен з наступним копіюванням цих імен у віджет Listbox списку множини А або В, в залежності від стану віджета Radiobutton, який вказує у який список відбувається копіювання імені.

5. Віджети Button для збереження у файл, зчитування з файлу та очищення множини А.

6. Віджети Button для збереження у файл, зчитування з файлу та очищення множини В.

### Вміст вікна №3

1. Віджети Listbox для відображення елементів множин А і В.

2. Матричне або графічне представлення відношення  $S$ , яке сформоване вашим алгоритмом

3. Матричне або графічне представлення відношення  $R$ , яке сформоване вашим алгоритмом з урахуванням несуперечливості з відношенням  $S$ .

### Вміст вікна №4

1. Матричне або графічне представлення відношення  $R \cup S$

2. Матричне або графічне представлення відношення  $R \cap S$



3. Матричне або графічне представлення відношення  $R \setminus S$
4. Матричне або графічне представлення відношення  $U \setminus R$
5. Матричне або графічне представлення відношення  $S^{-1}$

## 2.8. Варіанти для виконання лабораторної роботи

Варіант	Відношення 1	Відношення 2
1	aSb, якщо а мати b.	aRb, якщо а онука b.
2	aSb, якщо а батько b.	aRb, якщо а чоловік b.
3	aSb, якщо а кум b.	aRb, якщо а чоловік b.
4	aSb, якщо а хрещена мати b.	aRb, якщо а сестра b.
5	aSb, якщо а мати b.	aRb, якщо а дружина b.
6	aSb, якщо а дружина b.	aRb, якщо а кума b.
7	aSb, якщо а тесть b.	aRb, якщо а хрещений батько b.
8	aSb, якщо а чоловік b.	aRb, якщо а онук b.
9	aSb, якщо а мати b.	aRb, якщо а сестра b.
10	aSb, якщо а внук b.	aRb, якщо а батько b.
11	aSb, якщо а хрещена мати b.	aRb, якщо а теща b.
12	aSb, якщо а сват b.	aRb, якщо а хрещений батько b.
13	aSb, якщо а хрещений батько b.	aRb, якщо а свояк b.
14	aSb, якщо а теща b.	aRb, якщо а дружина b.
15	aSb, якщо а дружина b.	aRb, якщо а сестра b.
16	aSb, якщо а свекор b.	aRb, якщо а чоловік b.
17	aSb, якщо а батько b.	aRb, якщо а свекор b.
18	aSb, якщо а мати b.	aRb, якщо а свекруха b.
19	aSb, якщо а внучка b.	aRb, якщо а дружина b.
20	aSb, якщо а чоловік b.	aRb, якщо а брат b.
21	aSb, якщо а хрещена мати b.	aRb, якщо а кума b.
22	aSb, якщо а брат b.	aRb, якщо а батько b.
23	aSb, якщо а мати b.	aRb, якщо а теща b.
24	aSb, якщо а внучка b.	aRb, якщо а хрещена мати b.
25	aSb, якщо а хрещений батько b.	aRb, якщо а свекор b.
26	aSb, якщо а дружина b.	aRb, якщо а свекруха b.
27	aSb, якщо а кум b.	aRb, якщо а хрещений батько b.
28	aSb, якщо а хрещена мати b.	aRb, якщо а свекруха b.
29	aSb, якщо а чоловік b.	aRb, якщо а тесть b.
30	aSb, якщо а батько b.	aRb, якщо а тесть b.

### Лабораторна робота № 3

**Тема:** «Графи. Способи представлення графів. Остовні дерева. Пошук найкоротших шляхів».

**Мета роботи:** Вивчення властивостей графів, способів їх представлення та основних алгоритмів на графах.

**Завдання:** створити програму, яка реалізує один з алгоритмів на графах.

#### Теоретичні основи

### 3.1. ОСНОВНІ ОЗНАЧЕННЯ

Останнім часом теорія графів стала простим, доступним і потужним засобом вирішення завдань, що відносяться до широкого кола проблем. Це проблеми проектування інтегральних схем і схем управління, дослідження автоматів, логічних ланцюгів, блок-схем програм, економіки та статистики, хімії та біології, теорії розкладів і дискретної оптимізації.

**Граф**  $G$  задають множиною точок або вершин  $x_1, x_2, \dots, x_n$  (яку позначають через  $X$ ) і множиною ліній або ребер  $a_1, a_2, \dots$ , (яку позначають символом  $A$ ), що з'єднують між собою всі або частину цих точок. Таким чином, граф  $G$  повністю задають (і позначають) парою  $(X, A)$ .

Якщо ребра з множини  $A$  орієнтовані, що зазвичай показується стрілкою, то їх називають дугами, і граф з такими ребрами називають орієнтованим графом (рис. 3.1 (а)). Якщо ребра не мають орієнтації, то граф називають неорієнтованим (рис. 3.1 (б)).

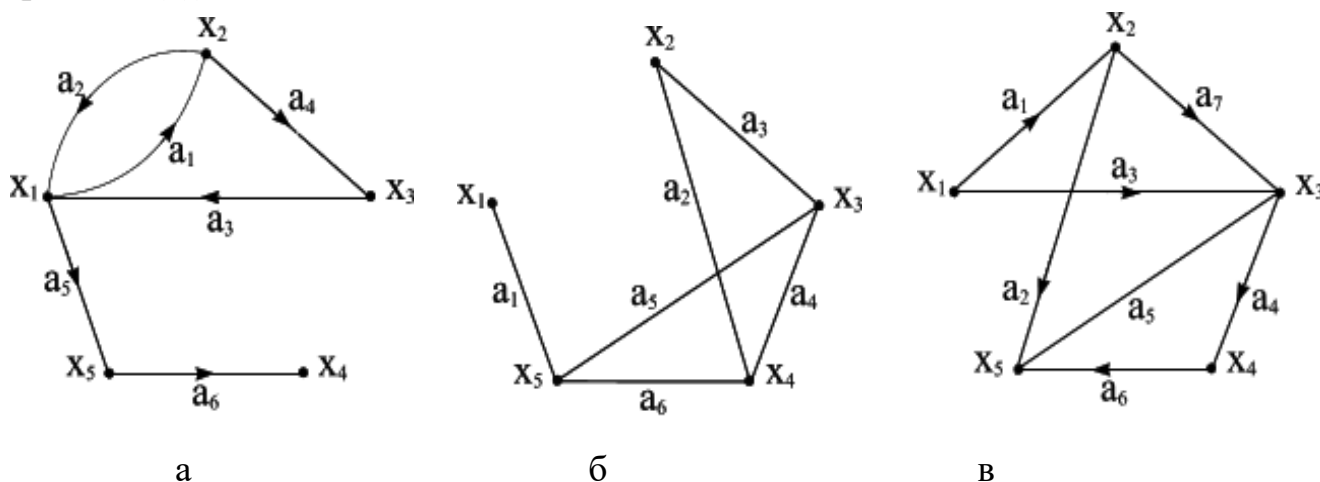


Рис. 3.1 (а) – орієнтований граф; (б) – неорієнтований граф; (в) – змішаний граф

Якщо дугу позначають впорядкованою парою, що складається з початкової та кінцевої вершин (тобто двома кінцевими вершинами дуги), її напрямок передбачається заданим від першої вершини до другої. Так, наприклад, на рис. 3.1 (а) позначення  $(x_1, x_2)$  відноситься до дуги  $a_1$ , а  $(x_2, x_1)$  – до дуги  $a_2$ .

Інший, вживаний частіше, опис орієнтованого графа  $G$  полягає у задаванні множини вершин  $X$  і відповідності  $\Gamma$ , яка показує, як між собою пов'язані вершини. Відповідність  $\Gamma$  називають відображенням множини  $X$  в  $X$ , а граф в цьому випадку позначають парою  $G = (X, \Gamma)$ .

Для графа на рис. 4.1 (а) маємо  $\Gamma(x_1) = \{x_2, x_5\}$ , тобто вершини  $x_2$  і  $x_5$  є кінцевими вершинами дуг, у яких початковою вершиною є  $x_1$ .

$$\Gamma(x_2) = \{x_1, x_3\}, \Gamma(x_3) = \{x_1\}, \Gamma(x_4) = \emptyset - \text{порожня множина}, \Gamma(x_5) = \{x_4\}$$

У разі неорієнтованого графа або графа, що містить і дуги, і неорієнтовані ребра (див., наприклад, графи, зображені на рис. 3.1 (б) і рис. 3.1 (в)), передбачається, що відповідність  $\Gamma$  задає такий еквівалентний орієнтований граф, який отримуємо з початкового графа заміною кожного неорієнтованого ребра двома протилежно спрямованими дугами, що з'єднують ті ж самі вершини. Так, наприклад, для графа, наведеного на рис. 3.1 (б), маємо  $\Gamma(x_5) = \{x_1, x_3, x_4\}$ ,  $\Gamma(x_1) = \{x_5\}$  і ін.

Оскільки пряма відповідність або образ вершини  $\Gamma(x_i)$  є множиною таких  $x_j \in X$ , для яких в графі  $G$  існує дуга  $(x_i, x_j)$ , то через  $\Gamma^{-1}(x_i)$  природно позначити множину вершин  $x_k$ , для яких в  $G$  існує дуга  $(x_k, x_i)$ . Таку відповідність прийнято називати *зворотною відповідністю* або *прообразом* вершини. Для графа, зображеного на рис. 3.1(а), маємо

$$\Gamma^{-1}(x_1) = \{x_2, x_3\}, \Gamma^{-1}(x_2) = \{x_1\} \text{ і т. д.}$$

Цілком очевидно, що для неорієнтованого графа  $\Gamma^{-1}(x_i) = \Gamma(x_i)$  для всіх  $x_i \in X$ .

Коли відображення  $\Gamma$  діє не на одну вершину, а на множину вершин  $X_q = \{x_1, x_2, \dots, x_q\}$ , то під  $\Gamma(X_q)$  розуміють об'єднання  $\Gamma(x_1) \cup \Gamma(x_2) \cup \dots \cup \Gamma(x_q)$ , тобто  $\Gamma(X_q)$  є множиною таких вершин  $x_j \in X$ , що для кожної з них існує дуга  $(x_i, x_j)$  в  $G$ , де  $x_i \in X_q$ . Для графа, наведеного на рис. 3.1(а),

$$\Gamma(\{x_2, x_5\}) = \{x_1, x_3, x_4\} \text{ і } \Gamma(\{x_1, x_3\}) = \{x_2, x_5, x_1\}.$$

Відображення  $\Gamma(\Gamma(x_i))$  записують як  $\Gamma^2(x_i)$ . Аналогічно "потрійне" відображення  $\Gamma(\Gamma(\Gamma(x_i)))$  записують як  $\Gamma^3(x_i)$  і т. д. Для графа, показаного на рис. 3.1(а), маємо:

$$\begin{aligned} \Gamma^2(x_1) &= \Gamma(\Gamma(x_1)) = \Gamma(\{x_2, x_5\}) = \{x_1, x_3, x_4\}; \\ \Gamma^3(x_1) &= \Gamma(\Gamma^2(x_1)) = \Gamma(\{x_1, x_3, x_4\}) = \{x_2, x_5, x_1\} \text{ і т. д.} \end{aligned}$$

Аналогічно потрібно розуміти позначення  $\Gamma^{-2}(x_i)$ ,  $\Gamma^{-3}(x_i)$  і т. д.

Дуги  $a = (x_i, x_j)$ ,  $x_i \leftrightarrow x_j$ , що мають спільні кінцеві вершини, називають суміжними. Дві вершини  $x_i$  і  $x_j$  називають суміжними, якщо яка-небудь з двох дуг  $(x_i, x_j)$  і  $(x_j, x_i)$  або обидві одночасно присутні в графі. Так, наприклад, на рис. 3.2

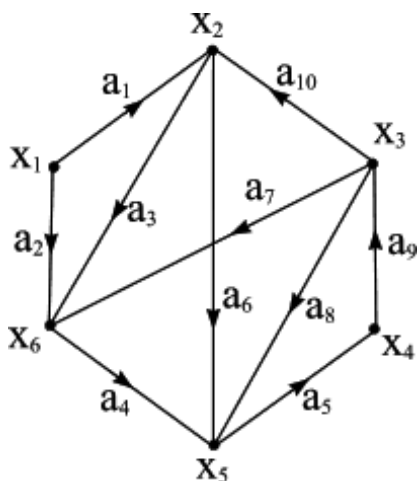


Рис. 3.2.

дуги  $a_1, a_{10}, a_3$  і  $a_6$ , як і вершини  $x_5$  і  $x_3$ , є суміжними, у той час як дуги  $a_1$  і  $a_5$  або вершини  $x_1$  і  $x_4$  не є суміжними.

Число дуг, які мають вершину  $x_i$  своєю початковою вершиною, називають **напівстепенем виходу** вершини  $x_i$ , і, аналогічно, число дуг, які мають  $x_i$  своєю кінцевою вершиною, називають напівстепенем входу вершини  $x_i$ .

Таким чином, на рис. 3.2 напівстепінь виходу вершини  $x_3$ , позначена через  $\deg^+(x_3)$ , дорівнює  $|\Gamma(x_3)|=3$ , і напівстепінь входу вершини  $x_3$ , позначена через  $\deg^-(x_3)$ , дорівнює  $|\Gamma^{-1}(x_3)|=1$ .

Очевидно, що сума напівстепенів входу всіх вершин графа, а також сума напівстепенів виходу всіх вершин дорівнюють загальному числу дуг графа  $G$ , тобто

$$\sum_{i=1}^n \deg^+(x_i) = \sum_{i=1}^n \deg^-(x_i) = m \quad (3.1)$$

де  $n$  – число вершин і  $m$  – число дуг графа  $G$ . Для неорієнтованого графа  $G=(X, \Gamma)$  степінь вершини  $x_i$  визначають аналогічно – за допомогою співвідношення  $\deg(x_i) = |\Gamma(x_i)| = |\Gamma^{-1}(x_i)|$ .

**Петлею** називають дугу, початкова та кінцева вершини якої збігаються.

Одним з найбільш важливих понять теорії графів є дерево. *Неорієнтованим деревом* називають зв'язний граф, що не має циклів.

*Суграфом* графа  $G$  є підграф  $G_p$ , що містить всі вершини початкового графа.

Якщо  $G=(X, A)$  – неорієнтований граф з  $n$  вершинами, то зв'язний суграф  $G_p$ , який не має циклів, називають *остовним деревом (остовом) графа  $G$* .

Для остовного дерева справедливе співвідношення:

$$G_p=(X_p, A_p) \subseteq G, \quad \text{где } X_p = X, A_p \subseteq A \quad (3.2)$$

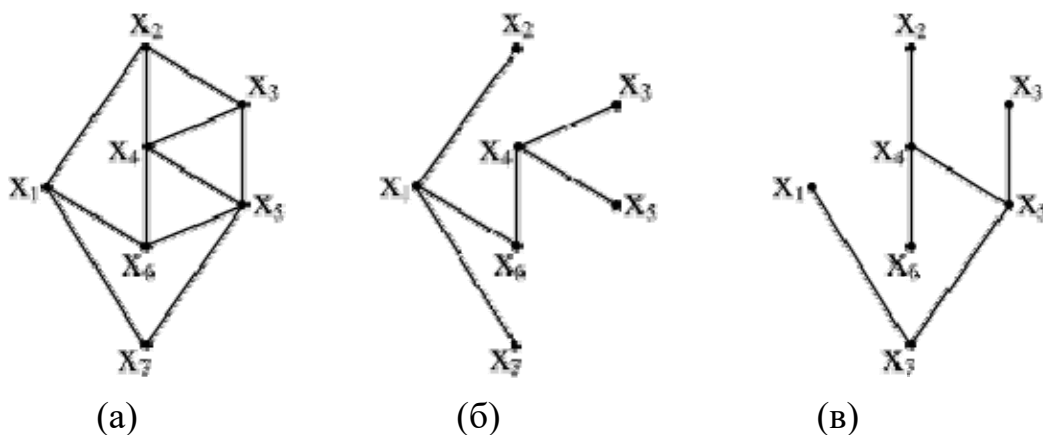


Рис. 3.3. Представлення графа у вигляді остовних дерев

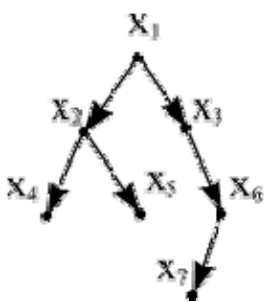
Легко довести, що остовне дерево має наступні властивості:

- 1) Остовне дерево графа з  $n$  вершинами має  $n-1$  ребро ( $|X_p|=|A_p|-1$ );
- 2) Існує єдиний шлях, що з'єднує будь-які дві вершини остова графа:  
 $\forall x_i, x_j \in X_p (i \neq j) \rightarrow \exists! \mu(x_i, x_j)$ .

Наприклад, якщо  $G$  – граф, показаний на рис. 3.3(а), то графі на рис. 3.3(б, в) є остовами графа  $G$ . Зі сформульованих вище визначень випливає, що остов графа  $G$  можна розглядати як мінімальний зв'язний остовний підграф графа  $G$ .

Поняття дерева як математичного об'єкта було вперше запропоновано Кірхгофом у зв'язку з визначенням фундаментальних циклів, застосовуваних при аналізі електричних ланцюгів. Приблизно десятьма роками пізніше Келі знову (незалежно від Кірхгофа) ввів поняття дерева і отримав більшу частину перших результатів у галузі дослідження властивостей дерев. Велику популярність здобула його знаменита теорема:

*Теорема Келі.* На графі з  $n$  вершинами можна побудувати  $n^{n-2}$  остови дерев.



*Орієнтоване дерево* є орієнтованим графом без циклів, в якому напівстепінь входу кожної вершини, за винятком однієї (вершини  $r$ ), дорівнює одиниці, а напівстепінь входу вершини  $r$  (названої коренем цього дерева) дорівнює нулю.

На рис. 3.4 показаний граф, який є орієнтованим деревом з коренем у вершині  $x_1$ . З наведеного визначення випливає, що орієнтоване дерево з  $n$  вершинами має  $n-1$  дуг і є зв'язним.

Неорієнтоване дерево можна перетворити в орієнтоване: треба взяти його довільну вершину як корінь і ребрам приписати таку орієнтацію, щоб кожна вершина з'єднувалася з коренем простим ланцюгом.

Рис. 3.4.

"Генеалогічне дерево", в якому вершини відповідають особам чоловічої статі, а дуги орієнтовані від батьків до дітей, є добре відомим прикладом орієнтованого дерева. Корінь в цьому дереві відповідає "засновнику" роду (особі, народженій раніше за інших).

**Шляхом** (або *орієнтованим маршрутом*) орієнтованого графа називають послідовність дуг, в якій кінцева вершина будь-якої дуги, відмінної від останньої, є початковою вершиною наступної. Так, на рис. 4.5 послідовності дуг  $\mu_1=\{a_6, a_5, a_9, a_8, a_4\}$ ,  $\mu_2=\{a_1, a_6, a_5, a_9\}$ ,  $\mu_3=\{a_1, a_6, a_5, a_9, a_{10}, a_6, a_4\}$  є шляхами.

**Орієнтованим ланцюгом** називають такий шлях, в якому кожна дуга використовується не більше одного разу. Так, наприклад, наведені вище шляхи  $\mu_1$  і  $\mu_2$  є орієнтованими ланцюгами, а шлях  $\mu_3$  не є таким, оскільки дуга  $a_6$  в ньому використовується двічі.

Маршрут є неорієнтованим "двійником" шляху, і це поняття розглядається в тих випадках, коли можна знехтувати спрямованістю дуг у графі.

Таким чином, маршрут є послідовністю ребер  $a_1, a_2, \dots, a_q$ , в якій кожне ребро  $a_i$ , за винятком, можливо, першого і останнього ребра, пов'язане з ребрами  $a_{i-1}$  і  $a_{i+1}$  своїми двома кінцевими вершинами.

Послідовності дуг на рис. 3.6  $\mu_4 = \{a_2, a_4, a_8, a_{10}\}$ ,  $\mu_5 = \{a_2, a_7, a_8, a_4, a_3\}$  і  $\mu_6 = \{a_{10}, a_7, a_4, a_8, a_7, a_2\}$  є маршрутами.

**Контуром** (простим ланцюгом) називають такий шлях (маршрут), в якому кожна вершина використовується не більше одного разу. Наприклад, шлях  $\mu_2$  є контуром, а шляхи  $\mu_1$  і  $\mu_3$  – ні. Очевидно, що контур є також ланцюгом, але зворотне твердження невірне. Наприклад, шлях  $\mu_1$  є ланцюгом, але не контуром, шлях  $\mu_2$  є ланцюгом і контуром, а шлях  $\mu_3$  не є ні ланцюгом, ні контуром.

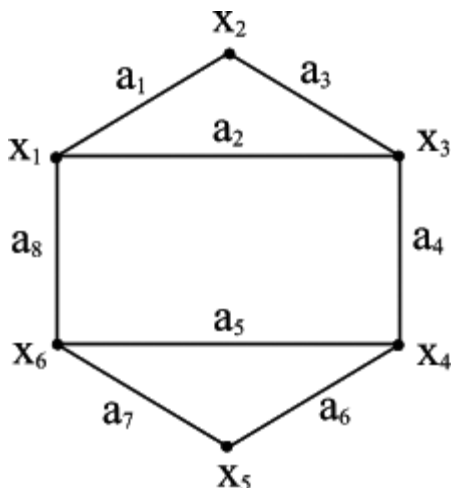


Рис. 3.5

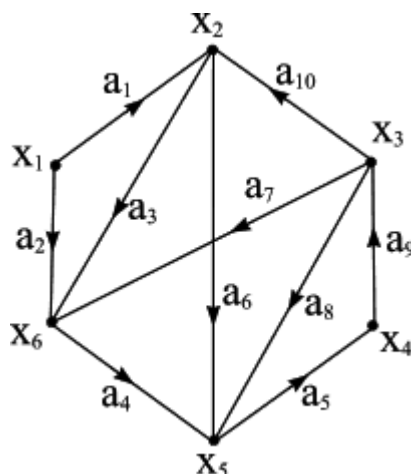


Рис. 3.6

Аналогічно визначають простий ланцюг у неорієнтованих графах. Так, наприклад, маршрут  $\mu_4$  є простим ланцюгом, маршрут  $\mu_5$  – ланцюг, а маршрут  $\mu_6$  не є ланцюгом.

Шлях або маршрут можна зображати також послідовністю вершин. Наприклад, шлях  $\mu_1$  можна представити так:  $\mu_1 = \{X_2, X_5, X_4, X_3, X_5, X_6\}$  і таке представлення часто виявляється більш корисним у тих випадках, коли здійснюється пошук контурів або простих ланцюгів.

Іноді дугам графа  $G$  співставляють (приписують) числа – дузі  $(x_i, x_j)$  ставлять у відповідність деяке число  $c_{ij}$ , назване **вагою**, або довжиною, або **вартістю** (ціною) дуги. Тоді граф  $G$  називають **зваженим**. Іноді ваги (числа  $v_i$ ) приписують вершинам  $x_i$  графа.

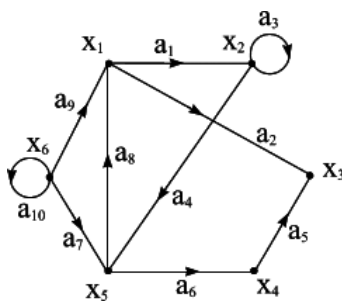
При розгляді шляху  $\mu$ , представленого послідовністю дуг  $(a_1, a_2, \dots, a_q)$ , за його вагу (або довжину, або вартість) приймають число  $L(\mu)$ , яке дорівнює сумі ваг всіх дуг, що входять до  $\mu$ , тобто

$$L(\mu) = \sum_{(x_i, x_j) \in \mu} c_{ij} \quad (3.3)$$

Таким чином, коли слова "довжина", "вартість", "ціна" і "вага" застосовують до дуг, то вони еквівалентні за змістом, і в кожному конкретному випадку вибирають таке слово, яке краще відповідає змісту завдання. **Довжиною** (або **потужністю**) шляху називають кількість (число) дуг, що входять до нього.

## 3.2. МАТРИЧНІ ПРЕДСТАВЛЕННЯ

### 3.2.1. МАТРИЦЯ СУМІЖНОСТІ



Нехай дано граф  $G$ , його матрицю суміжності позначають через  $A=[a_{ij}]$  і визначають наступним чином:

$a_{ij}=1$ , якщо в  $G$  існує дуга  $(x_i, x_j)$ ,  
 $a_{ij}=0$ , якщо в  $G$  немає дуги  $(x_i, x_j)$ .

Таким чином, матриця суміжності графа, зображеного на рис. 3.7, має вигляд:

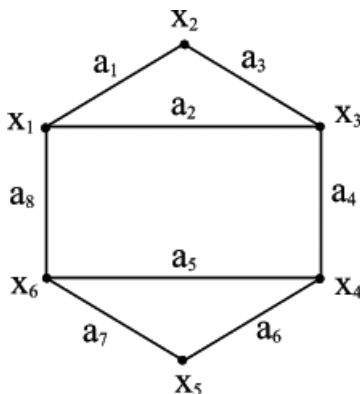
Рис. 3.7.

	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$
$x_1$	0	1	1	0	0	0
$x_2$	0	1	0	0	1	0
$x_3$	0	0	0	0	0	0
$x_4$	0	0	1	0	0	0
$x_5$	1	0	0	1	0	0
$x_6$	1	0	0	0	1	1

Матриця суміжності повністю визначає структуру графа. Наприклад, сума всіх елементів рядка  $x_i$  матриці дає напівстепені виходу вершини  $x_i$ , а сума елементів стовпця  $x_i$  - напівстепені входу вершини  $x_i$ . Множина стовпців, які мають 1 у рядку  $x_i$ , є множиною  $\Gamma(x_i)$ , а множина рядків, які мають 1 у стовпчику  $x_i$ , збігається з множиною  $\Gamma^{-1}(x_i)$ .

Петлі на графі є елементами, що мають 1 на головній діагоналі матриці, наприклад,  $a_{22}$ ,  $a_{66}$  для графа, зображеного на рис. 3.7.

У разі неорієнтованого графа матриця суміжності є симетричною відносно головної діагоналі (рис. 3.8).



	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$
$x_1$	0	1	1	0	0	1
$x_2$	1	0	1	0	0	0
$x_3$	1	1	0	1	0	0
$x_4$	0	0	1	0	1	1
$x_5$	0	0	0	1	0	1
$x_6$	1	0	0	1	1	0

Рис. 3.8

### 3.2.2. МАТРИЦЯ ІНЦИДЕНТНОСТІ

Нехай дано граф  $G$  з  $n$  вершинами і  $m$  дугами. Матриця інцидентності графа  $G$  позначається через  $\mathbf{B}=[b_{ij}]$  і є матрицею розмірності  $n \times m$ , яка визначається таким чином:

$b_{ij}=1$ , якщо  $x_i$  є початковою вершиною дуги  $a_j$ ;

$b_{ij}=-1$ , якщо  $x_i$  є кінцевою вершиною дуги  $a_j$ ;

$b_{ij}=0$ , якщо  $x_i$  не є кінцевою вершиною дуги  $a_j$ .

Для графа, наведеного на рисунку 3.7, матриця інцидентності має вигляд:

	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$	$a_7$	$a_8$	$a_9$	$a_{10}$
$x_1$	1	1	0	0	0	0	0	-1	-1	0
$x_2$	-1	0	$\pm 1$	1	0	0	0	0	0	0
$x_3$	0	-1	0	0	0	0	0	0	0	0
$x_4$	0	0	0	0	-1	-1	0	0	0	0
$x_5$	0	0	0	-1	1	1	-1	1	0	0
$x_6$	0	0	0	0	0	0	1	0	1	$\pm 1$

Оскільки кожна дуга інцидентна двом різним вершинам (за винятком випадку, коли дуга утворює петлю), то кожен стовпець містить один елемент, що дорівнює 1, і один – дорівнює -1. Петля в матриці інцидентності не має адекватного математичного подання (в програмній реалізації допустимо задавання одного елемента  $b_{ij}=1$ ).

Якщо  $G$  є неорієнтованим графом (рис. 3.8), то його матрицю інцидентності визначають наступним чином:

$b_{ij}=1$ , якщо  $x_i$  є кінцевою вершиною дуги  $a_j$ ;

$b_{ij}=0$ , якщо  $x_i$  не є кінцевою вершиною дуги  $a_j$ .

	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$	$a_7$	$a_8$
$x_1$	1	1	0	0	0	0	0	1
$x_2$	1	0	1	0	0	0	0	0
$x_3$	0	1	1	1	0	0	0	0
$x_4$	0	0	0	1	1	1	0	0
$x_5$	0	0	0	0	0	1	1	0
$x_6$	0	0	0	0	1	0	1	1

Матрицю інцидентності як спосіб задавання графів успішно застосовують при описі мультиграфів (графів, в яких суміжні вершини можуть з'єднуватися кількома паралельними дугами).



### 3.3. НАЙКОРОТШИЙ ОСТОВ ГРАФА

Розглянемо метод прямої побудови *найкоротших остовних дерев* у зваженому графі (в якому ваги приписані дугам). Найкоротше остовне дерево графа застосовують при прокладці доріг (газопроводів, ліній електропередач і т. д.), коли необхідно пов'язати  $n$  точок деякою мережею так, щоб загальна довжина "ліній зв'язку" була мінімальною. Якщо точки лежать на евклідовій площині, то їх можна вважати вершинами повного графа  $G$  з вагами дуг, що дорівнюють відповідним "прямолінійним" відстаням між кінцевими точками дуг. Якщо "розгалуження" доріг допускається тільки в заданих  $n$  точках, найкоротше остовне дерево графа  $G$  буде якраз необхідною мережею доріг, яка має найменшу вагу.

Розглянемо зважений зв'язний неорієнтований граф  $G=(X,A)$ ; вагу ребра  $(x_i, x_j)$  позначимо  $c_{ij}$ . З великого числа остовів графа потрібно знайти один, у якого сума ваг ребер найменша. Така задача виникає, наприклад, у тому випадку, коли вершини є клемами електричної мережі, які повинні бути з'єднані одна з одною за допомогою проводів найменшої загальної довжини (для зменшення рівня наведень). Інший приклад: вершини представляють міста, які потрібно пов'язати мережею трубопроводів; тоді найменшу загальну довжину труб, яка повинна бути використана для будівництва (за умови, що поза межами міст "розгалуження" трубопроводів не допускаються), визначають як найкоротший остов відповідного графа.

Завдання побудови найкоротшого остова графа є однією з небагатьох задач теорії графів, які можна вважати повністю вирішеними.

### 3.4. АЛГОРИТМ ПРИМА-КРАСКАЛА

Цей алгоритм породжує остовне дерево за допомогою розростання тільки одного піддерева, наприклад  $X_p$ , що містить більше однієї вершини. Піддерево поступово розростається за рахунок приєднання ребер  $(x_i, x_j)$ , де  $x_i \in X_p$  і  $x_j \notin X_p$ ; причому ребро, яке додається, повинно мати найменшу вагу  $c_{ij}$ . Процес продовжується доти, поки число ребер в  $A_p$  не стане рівним  $n-1$ . Тоді піддерево  $G_p=(X_p, A_p)$  буде необхідним остовним деревом. Вперше така операція була запропонована Примом і Краскалом (з різницею – в способі побудови дерева), тому даний алгоритм отримав назву Прима-Краскала.

Алгоритм починає роботу з включення в піддерево початкової вершини. Оскільки остовне дерево включає всі вершини графа  $G$ , то вибір початкової вершини не має принципового значення. Будемо кожній черговій вершині присвоювати позначку  $\beta(x_i)=1$ , якщо вершина  $x_i$  належить піддереву  $X_p$  і  $\beta(x_j)=0$  – в іншому випадку.

Алгоритм має вигляд:

*Крок 1.* Нехай  $X_p=\{x_1\}$ , де  $x_1$  - початкова вершина, і  $A_p=\emptyset$  ( $A_p$  є множиною ребер, що входять в остовне дерево). Вершині  $x_1$  присвоїти позначку  $\beta(x_1)=1$ . Для кожної вершини  $x_i \notin X_p$  присвоїти  $\beta(x_i)=0$ .

Крок 2. З усіх вершин  $x_j \in \Gamma(X_p)$ , для яких  $\beta(x_j)=0$ , знайти вершину  $x_j^*$  таку, що

$$c(x_i, x_j^*) = \min_{x_j \in \Gamma(X_p)} \{c(x_i, x_j)\}, \text{ де } x_i \in X_p \text{ и } x_j \notin X_p. \quad (3.4)$$

Крок 3. Оновити дані:  $X_p = X_p \cup \{x_j^*\}$ ;  $A_p = A_p \cup (x_i, x_j^*)$ . Присвоїти  $\beta(x_j^*)=1$ .

Крок 4. Якщо  $|X_p|=n$ , то зупинитися. Ребра в  $A_p$  утворюють найкоротший остов графа. Якщо  $|X_p|<n$ , то перейти до кроку 2.

### 3.5. КОНТРОЛЬНИЙ ПРИКЛАД

Для прикладу розглянемо граф, зображений на рис. 3.9 Знайдемо для нього найкоротше остовне дерево, з цією метою використовуючи розглянутий вище алгоритм Прима-Краскала. Позначимо множину суміжних вершин, що не входять в породжене піддерево, як  $\Gamma^*(X_p)$ . Таким чином, для всіх вершин, що входять у цю множину, оцінка  $\beta(x_j)=0, \forall x_j \in \Gamma^*(X_p)$ . Вектор  $B$  є множиною оцінок  $\beta(x_i)$  для всіх вершин графа  $G$ :  $\forall x_i \in X$ . Довжина породженого піддерева позначається як  $L$ .

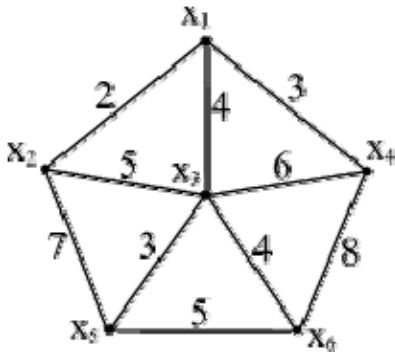


Рис. 3.9.

Ітерація 1:  $X_p=\{x_1\}$ ;  $A_p=\emptyset$ ;  $\Gamma^*(X_p)=\{x_2, x_3, x_4\}$ ;

$c(x_1, x_2^*)=2$ ;  $B=\{1,1,0,0,0,0\}$ ;  $L=2$ .

Ітерація 2:  $X_p=\{x_1, x_2\}$ ;  $A_p=\{(x_1, x_2)\}$ ;

$\Gamma^*(X_p)=\{x_3, x_4, x_5\}$ ;  $c(x_1, x_4^*)=3$ ;

$B=\{1,1,0,10,0\}$ ;  $L=2+3=5$ .

Ітерація 3:  $X_p=\{x_1, x_2, x_4\}$ ;  $A_p=\{(x_1, x_2); (x_1, x_4)\}$ ;

$\Gamma^*(X_p)=\{x_3, x_5, x_6\}$ ;  $c(x_1, x_3^*)=4$ ;

$B=\{1,1,1,1,0,0\}$ ;  $L=5+4$ .

Ітерація 4:  $X_p=\{x_1, x_2, x_3, x_4\}$ ;  $A_p=\{(x_1, x_2); (x_1, x_4); (x_1, x_3)\}$ ;

$\Gamma^*(X_p)=\{x_5, x_6\}$ ;  $c(x_3, x_5^*)=2$ ;  $B=\{1,1,1,1,1,0\}$ ;  $L=9+3=12$ .

Ітерація 5:  $X_p=\{x_1, x_2, x_3, x_4, x_5\}$ ;  $A_p=\{(x_1, x_2); (x_1, x_4); (x_1, x_3); (x_3, x_5)\}$ ;

$\Gamma^*(X_p)=\{x_6\}$ ;  $c(x_3, x_6^*)=4$ ;  $B=\{1,1,1,1,1,1\}$ ;  $L=12+4=16$ .

Задача розв'язана. Отримані множини вершин  $X_p$  і ребер  $A_p$  складають найкоротше остовне дерево:

$X_p=\{x_1, x_2, x_3, x_4, x_5, x_6\}$ ;  $A_p = \{(x_1, x_2); (x_1, x_4); (x_1, x_3); (x_3, x_5); (x_3, x_6)\}$ ;

Сумарна довжина найкоротшого остовного дерева  $L=16$ .

Результат розв'язання задачі представлений на рис. 3.10.

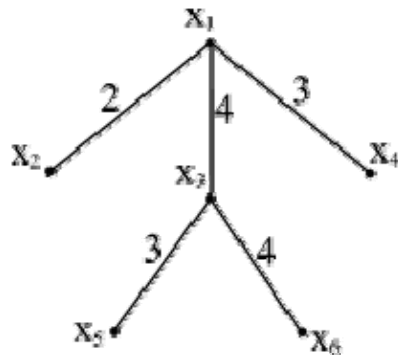


Рис. 3.10.

### 3.6. ЗАДАЧА ПРО НАЙКОРОТШИЙ ШЛЯХ

Нехай дано граф  $G=(X,\Gamma)$ , дугам якого приписані ваги (вартості), що задані матрицею  $\mathbf{C}=[c_{ij}]$ . Задача про найкоротший шлях полягає в знаходженні найкоротшого шляху від заданої початкової вершини (витоку)  $s$  до заданої кінцевої вершини (стоку)  $t$ , за умови, що такий шлях існує.

Знайти  $\mu(s,t)$  при  $L(\mu) \rightarrow \min$ ,  $s,t \in X$ ,  $t \in R(s)$ , де  $R(s)$  – множина, досяжна з вершини  $s$ .

У загальному випадку елементи  $c_{ij}$  матриці ваг  $\mathbf{C}$  можуть бути додатними, від'ємними або нулями. Єдине обмеження полягає в тому, щоб в  $G$  не було циклів з сумарною від'ємною вагою. Звідси випливає, що дуги (ребра) графа  $G$  не повинні мати від'ємні ваги.

Майже всі методи, що дозволяють вирішити задачу про найкоротший  $(s-t)$ -шлях, дають також (в процесі вирішення) і всі найкоротші шляхи від  $s$  до  $x_i (\forall x_i \in X)$ . Таким чином, вони дозволяють вирішити задачу з невеликими додатковими обчислювальними витратами.

Допускається, що матриця ваг  $\mathbf{C}$  не задовольняє умову трикутника, тобто не обов'язково  $c_{ij} \leq c_{ik} + c_{kj}$  для всіх  $i, j$  і  $k$ .

Якщо в графі  $G$  дуга  $(x_i, x_j)$  відсутня, то її вага дорівнює  $\infty$ .

Ряд задач, наприклад, задача знаходження в графах шляхів з максимальною надійністю і з максимальною пропускнуою здатністю, пов'язані із задачею про найкоротший шлях, хоча в них характеристика шляху (скажімо, вага) є не сумою, а деякою іншою функцією характеристик (ваг) дуг, які утворюють шлях. Такі задачі можна переформулювати як задачі про найкоротший шлях і вирішувати їх відповідним чином.

Існує безліч методів вирішення даної задачі, що відрізняються областю застосування і трудомісткістю (Дейкстри, Флойда, динамічного програмування). Серед них велике поширення одержали спеціальні алгоритми, що застосовуються при вирішенні окремих задач, і мають меншу трудомісткість. Ці окремі випадки зустрічаються на практиці досить часто (наприклад, коли  $c_{ij}$  є відстанями), так що розгляд цих спеціальних алгоритмів виправданий.

### 3.6.1. АЛГОРИТМ ДЕЙКСТРИ

Алгоритм Дейкстри знаходить найкоротший шлях для випадку  $c_{ij} \geq 0$ .

1. Вершинам приписують **тимчасові позначки**. Нехай  $l(v_i)$  – позначка вершини  $v_i$ .
2. Кожна позначка вершини дає **верхню границю довжини шляху** від  $S$  до цієї вершини.
3. Величини цих **позначок зменшуються ітераційно**.
4. На кожному кроці ітерації одна з тимчасових позначок **стає постійною**.
5. Величина цієї позначки є точною довжиною найкоротшого шляху від  $S$  до розглянутої вершини.

#### Теоретичний опис алгоритму Дейкстри

##### Присвоєння початкових значень

*Крок 1.* Встановити  $l(s) := 0$  і вважати цю позначку постійною. Встановити  $l(v_i) := \infty$  для всіх  $v_i \neq s$  і вважати ці позначки тимчасовими. Встановити  $p = s$

##### Відновлення позначок

*Крок 2.* Для всіх  $v_i \in \Gamma(p)$ , позначки яких тимчасові, змінити позначки у відповідності з наступним виразом:

$$l(v_i) \leftarrow \min[l(v_i), l(p) + c(p, v_i)]$$

##### Перетворення позначки в постійну

*Крок 3.* Серед вершин з тимчасовими позначками знайти таку, для якої

$$l(v_i^*) = \min l(v_i), v_i \in \Gamma(p)$$

*Крок 4.* Вважати позначку  $l(v_i^*)$  постійною і встановити  $p = v_i^*$ .

*Крок 5.* Коли треба знайти шлях від  $S$  до  $t$ . Якщо  $p = t$ , то  $l(p)$  є довжиною найкоротшого шляху від вершини  $S$  до вершини  $t$ . Останов.

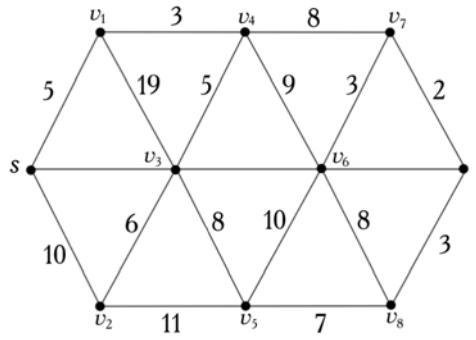
*Крок 6.* Якщо  $p \neq t$ , перейти до кроку 2.

*Крок 7.* **Якщо потрібно знайти шляхи від  $S$  до всіх інших вершин.** Якщо всі вершини відзначені як постійні, то ці позначки дають довжини найкоротших шляхів. Останов.

*Крок 8.* Якщо деякі позначки є змінними, перейти до кроку 2.

#### Приклад ручного розв'язку завдання

Розглянемо граф, зображений на рисунку, де кожне неорієнтоване ребро розглядається як пара протилежно орієнтованих дуг рівної ваги. Ця вага виписана на малюнку, що зображує граф, поруч із ребром.



Потрібно знайти найкоротші шляхи від вершини  $s$  до всіх інших вершин. Для цього використовуємо алгоритм Дейкстри. Постійні позначки відзначають знаком  $+$ , інші позначки є тимчасовими.

### Схема застосування алгоритму

Крок 1.  $l(s) = 0^+, l(v_i) = \infty, i = 1, \dots, 8, p = s$ .

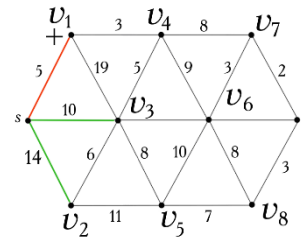
#### Перша ітерація

Крок 2.  $\Gamma(s) = \{v_1, v_2, v_3\}$  – усі позначки тимчасові.

$$l(v_1) = \min[\infty, 0^+ + 5] = 5,$$

$$l(v_2) = \min[\infty, 0^+ + 14] = 14,$$

$$l(v_3) = \min[\infty, 0^+ + 10] = 10.$$



Крок 3.  $l(v_1) = \min_{i=1,2,3} l(v_i) = 5$ .

Крок 4.  $l(v_1) = 5^+ - v_1$  одержує постійну позначку;  $p = v_1$ .

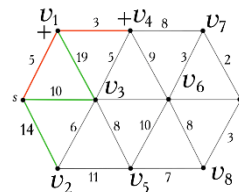
Крок 5. Не всі вершини мають постійні позначки, тому переходимо до кроку 2.

#### Друга ітерація

Крок 2.  $\Gamma(p) = \Gamma(v_1) = \{s, v_3, v_4\}$ . Вершина  $s$  має постійну позначку;

$$l(v_3) = \min[10, 5^+ + 19] = 10,$$

$$l(v_4) = \min[\infty, 5^+ + 3] = 8.$$



Крок 3.  $l(v_4) = \min_{i=3,4} l(v_i)$ .

Крок 4. Вершина  $v_4$  одержує постійну мітку:  $l(v_4) = 8^+$ ;  $p = v_4$ .

Крок 5. Не всі вершини мають постійні позначки, тому переходимо до кроку 2.

### Третя ітерація

Крок 2.  $\Gamma(p) = \Gamma(v_4) = \{v_1, v_3, v_6, v_7\}$ . Вершина  $v_1$  має постійну позначку;

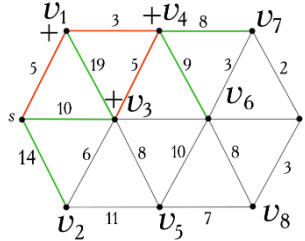
$$l(v_3) = \min[10, 8^+ + 5] = 10,$$

$$l(v_7) = \min[\infty, 8^+ + 8] = 16,$$

$$l(v_6) = \min[\infty, 8^+ + 9] = 17.$$

$$l(v_3) = \min_{i=3,6,7} l(v_i) = 10$$

Крок 3.



Крок 4. Вершина  $v_3$  одержує постійну мітку:  $l(v_3) = 10^+$ ;  $p = v_3$ .

Крок 5. Не всі вершини мають постійні позначки, тому переходимо до кроку 2.

### Четверта ітерація

Крок 2.  $\Gamma(p) = \Gamma(v_3) = \{s, v_1, v_2, v_4, v_5, v_6\}$ . Вершини  $s, v_1, v_4$  мають постійні позначки;

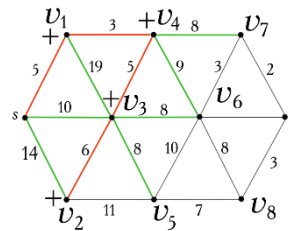
$$l(v_2) = \min[14, 10^+ + 6] = 14,$$

$$l(v_5) = \min[\infty, 10^+ + 8] = 18,$$

$$l(v_6) = \min[17, 10^+ + 8] = 17.$$

$$l(v_2) = \min_{i=2,5,6} l(v_i) = 14$$

Крок 3.



Крок 4. Вершина  $v_2$  одержує постійну мітку:  $l(v_2) = 14^+$ ;  $p = v_2$ .

Крок 5. Не всі вершини мають постійні позначки, тому переходимо до кроку 2.

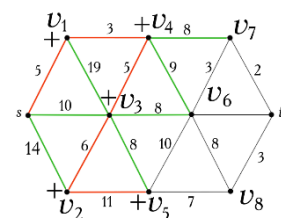
### П'ята ітерація

Крок 2.  $\Gamma(p) = \Gamma(v_2) = \{s, v_3, v_5\}$ . Вершини  $s, v_3$  мають постійні позначки;

$$l(v_5) = \min[18, 14^+ + 11] = 18.$$

$$l(v_5) = \min_{i=5} l(v_i) = 18$$

Крок 3.



Крок 4. Вершина  $v_5$  одержує постійну мітку:  $l(v_5) = 18^+$ ;  $p = v_5$ .

Крок 5. Не всі вершини мають постійні позначки, тому переходимо до кроку 2.

### Шоста ітерація

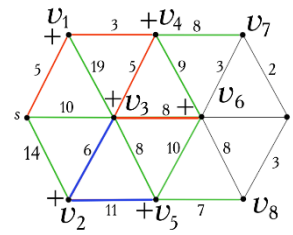
**Крок 2.**  $\Gamma(p) = \Gamma(v_5) = \{v_2, v_3, v_6, v_8\}$ . Вершини  $v_2, v_3$  мають постійні позначки;

$$l(v_6) = \min[17, 18^+ + 10] = 17,$$

$$l(v_8) = \min[\infty, 18^+ + 7] = 25.$$

$$l(v_6) = \min_{i=6,8} l(v_i) = 17$$

Крок 3.



Крок 4. Вершина  $v_6$  одержує постійну мітку:  $l(v_6) = 17^+$ ;  $p = v_6$ .

Крок 5. Не всі вершини мають постійні позначки, тому переходимо до кроку 2.

### Сьома ітерація

**Крок 2.**  $\Gamma(p) = \Gamma(v_6) = \{v_3, v_4, v_5, v_7, v_8, t\}$ . Вершини  $v_3, v_4, v_5$  мають постійні позначки;

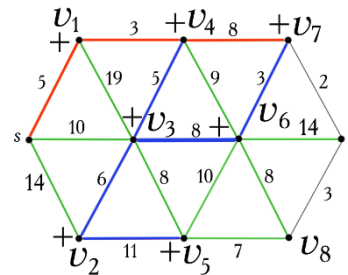
$$l(v_7) = \min[16, 17^+ + 3] = 16,$$

$$l(v_8) = \min[25, 17^+ + 8] = 25,$$

$$l(t) = \min[\infty, 17^+ + 14] = 31.$$

$$l(v_7) = \min_{i=7,8,t} l(v_i) = 16$$

Крок 3.



Крок 4. Вершина  $v_7$  одержує постійну мітку:  $l(v_7) = 16^+$ ;  $p = v_7$ .

Крок 5. Не всі вершини мають постійні позначки, тому переходимо до кроку 2.

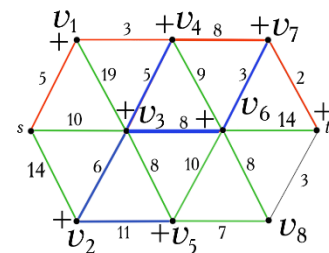
### Восьма ітерація

**Крок 2.**  $\Gamma(p) = \Gamma(v_7) = \{v_4, v_6, t\}$ . Вершини  $v_4, v_6$  мають постійні позначки;

$$l(t) = \min[31, 16^+ + 2] = 18.$$

$$l(v_t) = \min_{i=t} l(v_i) = 18$$

Крок 3.



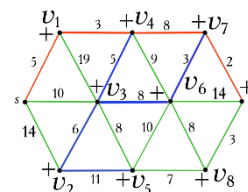
Крок 4. Вершина  $t$  одержує постійну мітку:  $l(t) = 18^+$ ;  $p = t$ .

Крок 5. Не всі вершини мають постійні позначки, тому переходимо до кроку 2.

## Дев'ята ітерація

Крок 2.  $\Gamma(p) = \Gamma(t) = \{v_6, v_7, v_8\}$ . Вершини  $v_6, v_7$  мають постійні позначки;  
 $l(v_8) = \min[25, 18^+ + 3] = 21$ .

Крок 3.  $l(v_8) = \min_{i=8} l(v_i) = 21$ .



Крок 4. Вершина  $v_8$  одержує постійну мітку:  $l(v_8) = 21^+$ ;  $p = v_8$ .

Крок 5. Позначки всіх вершин постійні. Останов.

## Зауваження

Як тільки всі позначки вершин графа стають постійними, тобто знайдені довжини найкоротших шляхів від вершини  $s$  до будь-якої іншої вершини, самі шляхи можна одержати, використовуючи рівність

$$l(v'_i) + c(v'_i, v_i) = l(v_i).$$

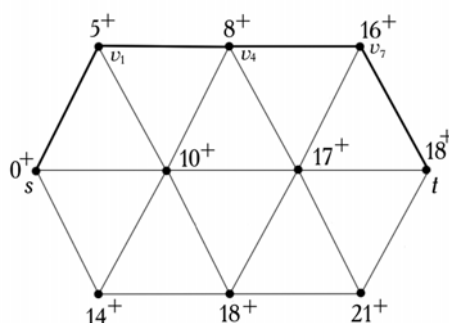
Тут  $c(v'_i, v_i)$  – вага дуги, що з'єднує вершину  $v'_i$  з вершиною  $v_i$ . Ця рівність дає можливість для кожної вершини  $v_i$  знайти попередню їй вершину  $v'_i$  й відновити весь шлях від  $s$  до  $v_i$ .

У розглянутому прикладі найкоротший шлях від вершини  $s$  до вершини  $t$  можна відновити з рівностей:

$$\begin{aligned} l(t) &= l(v_6) + c(v_6, t), \\ l(v_6) &= l(v_4) + c(v_4, v_7), \\ l(v_4) &= l(v_1) + c(v_1, v_4), \\ l(v_1) &= l(s) + c(s, v_1), \end{aligned}$$

отриманих в 7-й, 2-й і 1-й ітераціях.

Йому відповідає послідовність вершин  $s, v_1, v_4, v_7, t$ .





## Опис машинного алгоритму Дейкстри

Алгоритм Дейкстри знаходить відстань від однієї вершини (дамо їй номер 0) до всіх інших, використовуючи кількість операцій порядку  $O(n^2)$ . Всі ваги ребер у графі додатні.

Базовою структурою даних для цього алгоритму є список  $g[0..n-1][0..n-1]$ , кожен елемент якого  $g[i][j]$  містить вагу ребра, що з'єднує вершину  $i$  з вершиною  $j$  у випадку, якщо таке ребро існує, або  $\text{inf}$ , якщо відповідного ребра в графі не існує.

На кожній ітерації позначають певну підмножину вершин. Задамо список з логічними змінними  $\text{mark}[0 \dots n-1]$ . Індекс цього списку відповідає номеру вершини. Значення кожного елементу списку дорівнює «True», якщо вершина позначена і дорівнює «False», якщо вершина не позначена.

Задамо також список  $d[0 \dots n-1]$  такого ж розміру. В цьому списку для кожної вершини буде зберігатися довжина найкоротшого шляху, що проходить тільки по помічених вершинах, як по «пересадочних». Також підтримується інваріант того, що для помічених вершин довжина, яка зазначена в  $d$ , і є відповідь. Спочатку позначена тільки вершина 0, а  $g[0][j]$  дорівнює  $x$ , якщо вершина 0 з'єднана ребром вагою  $x$  з вершиною  $j$ , або  $\text{inf}$ , якщо їх не з'єднує ребро, і дорівнює 0, якщо  $j = 0$ .

На кожній ітерації ми знаходимо вершину, з найменшим значенням в  $d$  серед непомічених, нехай це вершина  $v$ . Тоді значення  $d[v]$  є відповіддю для  $v$ .  
**Доведення.**

Нехай найкоротший шлях до  $v$  з 0 проходить не тільки по помічених вершинах як по «пересадочних», і при цьому він коротший за  $d[v]$ . Візьмемо першу непомічену вершину, яка зустрілася на цьому шляху, назовемо її  $u$ . Довжина пройденої частини шляху (від 0 до  $u$ ) –  $d[u]$ . Тоді  $\text{len} \geq d[u]$ , де  $\text{len}$  – довжина найкоротшого шляху з 0 до  $v$  (оскільки від'ємних ребер немає), але за нашим припущенням  $\text{len}$  менше  $d[v]$ . Отже,  $d[v] > \text{len} \geq d[u]$ . Але тоді  $v$  не підходить під свій опис – у неї не найменше значення  $d[v]$  серед непомічених. Виникло протиріччя. Тому  $\text{len}$  є довжиною найкоротшого шляху з 0 до  $v$ .

Тому помічаємо вершину  $v$  і перераховуємо  $d$ . Так робимо, поки всі вершини не стануть поміченими, і  $d$  не стане відповіддю на завдання.

Загальна кількість операцій:  $n$  ітерацій по  $n$  ітерацій (на пошук вершини  $v$ ), отже, разом  $O(n^2)$  операцій.

### Псевдокод алгоритму

Прочитати  $g$  //  $g[0 \dots n-1][0 \dots n-1]$  – список, в якому зберігаємо ваги ребер,  $g[i][j] = \text{inf}$ , якщо ребро між  $i$  та  $j$  не існує

$d = g$

$d[0] = 0$

```

mark[0] = True
for i = 1 ... n - 1
    mark[i] = False
for i = 1 ... n - 1
    v = -1
    for i = 0 ... n - 1
        if (not mark[i]) and ((v == -1) or (d[v] > d[i]))
            v = i
    mark[v] = True
    for i = 0 ... n - 1
        if d[i] > d[v] + g[v][i]
            d[i] = d[v] + g[v][i]
вивести d

```

### Приклад програми за алгоритмом Дейкстри

```

def dijkstra(G,s):
    print "StartVertice:",s
    # хеш-таблиця вершин, для яких побудовано найкоротші шляхи,
    (вершина:вага)
    Visited={}
    # хеш-таблиця для вершин, для яких ще не побудовані шлях,
    (вершина:вага)
    ToVisit={s:0}
    Paths = {s:[s]} # хеш-таблиця (вершина:найкоротший шлях)
    while ToVisit: # поки є вершини, для яких не побудовно
найкоротший шлях
        print Visited, ToVisit, "----->",
        v=argmin(ToVisit) # вибираємо найближчу
        Visited[v]=ToVisit[v]; del ToVisit[v];# для неї шлях
знайдено
        for w in G.neighbors(v): # для всіх сусідів вершини $v$
            if (w not in Visited): # до яких ще не знайшли
найкоротший шлях
                vwLength = Visited[v] +
G.get_edge(v,w)#оновлюємо шляхи
                if (w not in ToVisit) or (vwLength < ToVisit[w]):
                    ToVisit[w] = vwLength
                    Paths[w] = Paths[v]+[w]
        print Visited, ToVisit
    print Visited, Paths
    return (Visited,Paths)

```

В алгоритмі Дейкстри ми ітераційно підтримуємо дві множини вершин:

*Visited* – множина вершин, до яких ми вже знайшли найкоротший шлях, асоційований з вагою найкоротших шляхів від стартової вершини до них.

*ToVisit* – множина вершин, які досяжні через одне ребро з множини вершин *Visited*, які асоційовані з верхніми оцінками ваги шляху до них.

На кожній ітерації ми вибираємо з досяжних вершин вершину *v*, найближчу до стартової вершини *s*, і переносимо її з множини *ToVisit* в множину

Visited, збільшуємо множину «кандидатів» ToVisit її сусідами, і перераховуємо верхню оцінку віддаленості вершин з ToVisit до вершини  $s$ .

В результаті виконання алгоритму роздруковують зміну хеш-таблиць Visited і ToVisit на кожній ітерації, а в кінці роздруковують вхідний граф, де суцільними лініями намальовані наявні ребра з асоційованими довжинами, а пунктиром – знайдені найкоротші шляхи.

Важливо, що алгоритм працездатний тільки в разі додатних відстаней.

### 3.6.2. АЛГОРИТМ ФОРДА-БЕЛЛМАНА ЗНАХОДЖЕННЯ МІНІМАЛЬНОГО ШЛЯХУ

Передбачається, що орієнтований граф не містить контурів від'ємної довжини.

*Алгоритм 1. (Алгоритм Форда – Беллмана)*

Основними величинами цього алгоритму є величини  $\lambda_i(k)$ , де  $i = 1, 2, \dots, n$  ( $n$  – число вершин графа);  $k = 1, 2, \dots, n - 1$ .

Для фіксованих  $i$  і  $k$  величина  $\lambda_i(k)$  дорівнює довжині мінімального шляху, що веде із заданої початкової вершини  $v_1$  у вершину  $v_i$  і складається не більше, ніж з  $k$  дуг.

*Крок 1.* Установка початкових умов. Ввести число вершин графа  $n$  і матрицю ваг  $C = |c_{ij}|$ .

*Крок 2.* Встановити  $k = 0$ . Встановити  $\lambda_i(0) = \infty$  для всіх вершин, крім  $v_1$ ; встановити  $\lambda_1(0) = 0$ .

*Крок 3.* У циклі по  $k$ ,  $k = 1, 2, \dots, n - 1$ , кожній вершині  $v_i$  на  $k$ -му кроці приписати індекс  $\lambda_i(k)$  за наступним правилом:

$$\lambda_i(k) = \min_{1 \leq j \leq n} \{ \lambda_j(k-1) + c_{ji} \} \quad (3.5)$$

для всіх вершин, крім  $v_1$ , встановити  $\lambda_1(k) = 0$ .

У результаті роботи алгоритму формується таблиця індексів

$$\lambda_i(k), \quad i = 1, 2, \dots, n; \quad k = 0, 1, 2, \dots, n - 1.$$

При цьому  $\lambda_i(k)$  визначає довжину мінімального шляху з першої вершини в  $i$ -у, що містить не більше, ніж  $k$  дуг.

*Крок 5.* Відновлення мінімального шляху. Для будь-якої вершини  $v_s$  попередня їй вершина  $v_r$  визначається з співвідношення:

$$\lambda_r(n-2) + c_{rs} = \lambda_s(n-1), \quad v_r \in G^{-1}(v_s), \quad (3.6)$$

де  $G^{-1}(v_s)$  – прообраз вершини  $v_s$ .

Для знайденої вершини  $v_r$  попередня їй вершина  $v_q$  визначається зі співвідношення:

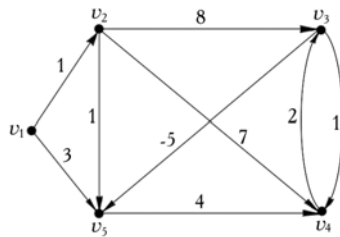
$$\lambda_q(n-3) + c_{qr} = \lambda_r(n-2), v_q \in G^{-1}(v_r),$$

де  $G^{-1}(v_r)$  – прообраз вершини  $v_r$ , і т. д.

Послідовно застосовуючи це співвідношення, починаючи від останньої вершини  $v_i$ , знайдемо мінімальний шлях.

### Приклад ручного розв'язку

За допомогою алгоритму Форда-Беллмана знайдемо мінімальний шлях з вершини  $v_1$  у вершину  $v_3$  на графі, зображеному на рисунку.



Значення індексів  $\lambda_i(k)$  будемо заносити в таблицю індексів (табл. 3.1).

Крок 1. Введемо число вершин графа  $n = 5$ . Матриця ваг цього графа має вигляд:

$$C = \begin{pmatrix} \infty & 1 & \infty & \infty & 3 \\ \infty & \infty & 8 & 7 & 1 \\ \infty & \infty & \infty & 1 & -5 \\ \infty & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & 4 & \infty \end{pmatrix}$$

Крок 2. Встановимо  $k = 0$ ,

$$\lambda_1(0) = 0, \lambda_2(0) = \lambda_3(0) = \lambda_4(0) = \lambda_5(0) = \infty$$

Ці значення занесемо в перший стовпець табл. 1.

Крок 3.  $k = 1$ .  $\lambda_1(0) = 0$ .

$$\lambda_1(1) = 0.$$

Рівність (4.5) для  $k = 1$  має вигляд:

$$\lambda_i(1) = \min_{1 \leq j \leq 5} \{ \lambda_j(0) + c_{ji} \}$$

$$\begin{aligned} \lambda_2(1) &= \min \{ \lambda_1(0) + c_{12}; \lambda_2(0) + c_{22}; \lambda_3(0) + c_{32}; \lambda_4(0) + c_{42}; \lambda_5(0) + c_{52} \} = \\ &= \min \{ 0 + 1; \infty + \infty; \infty + \infty; \infty + \infty; \infty + \infty \} = 1. \end{aligned}$$

$$\begin{aligned} \lambda_3(1) &= \min \{ \lambda_1(0) + c_{13}; \lambda_2(0) + c_{23}; \lambda_3(0) + c_{33}; \lambda_4(0) + c_{43}; \lambda_5(0) + c_{53} \} = \\ &= \min \{ 0 + \infty; \infty + 8; \infty + \infty; \infty + 2; \infty + \infty \} = \infty. \end{aligned}$$

$$\lambda_4(1) = \min \{ \lambda_1(0) + c_{14}; \lambda_2(0) + c_{24}; \lambda_3(0) + c_{34}; \lambda_4(0) + c_{44}; \lambda_5(0) + c_{54} \} =$$

$$= \min \{ 0 + \infty; \infty + 7; \infty + 1; \infty + \infty; \infty + 4 \} = \infty.$$

$$\begin{aligned} \lambda_5(1) &= \min \{ \lambda_1(0) + c_{15}; \lambda_2(0) + c_{25}; \lambda_3(0) + c_{35}; \lambda_4(0) + c_{45}; \lambda_5(0) + c_{55} \} = \\ &= \min \{ 0 + 3; \infty + 1; \infty - 5; \infty + \infty; \infty + \infty \} = 3. \end{aligned}$$

Отримані значення  $\lambda_i(1)$  занесемо в другий стовпець таблиці. Переконаємося, що другий стовпець, починаючи із другого елемента, збігається з першим рядком матриці ваг, що легко пояснюється змістом величин  $\lambda_i(1)$ , які дорівнюють довжині мінімального шляху з першої вершини в  $i$ -у, яка містить не більше однієї дуги.

$$k = 2. \lambda_1(2) = 0.$$

Рівність (4.5) для  $k = 2$  має вигляд:

$$\lambda_i(2) = \min_{1 \leq j \leq 5} \{ \lambda_j(1) + c_{ji} \}$$

$$\lambda_2(2) = \min \{ 0 + 1; 1 + \infty; \infty + \infty; \infty + \infty; 3 + \infty \} = 1.$$

$$\lambda_3(2) = \min \{ 0 + \infty; 1 + 8; \infty + \infty; \infty + 2; 3 + \infty \} = 9.$$

$$\lambda_4(2) = \min \{ 0 + \infty; 1 + 7; \infty + 1; \infty + \infty; 3 + 4 \} = 7.$$

$$\lambda_5(2) = \min \{ 0 + 3; 1 + 1; \infty - 5; \infty + \infty; 3 + \infty \} = 2.$$

Отримані значення  $\lambda_i(2)$  занесемо в третій стовпець таблиці. Величини  $\lambda_i(2)$  дорівнюють довжині мінімального шляху з першої вершини в  $i$ -ю, що містить не більш двох дуг.

$$k = 3. \lambda_1(3) = 0.$$

Рівність (1) для  $k = 3$  має вигляд:

$$\lambda_i(3) = \min_{1 \leq j \leq 5} \{ \lambda_j(2) + c_{ji} \}$$

$$\lambda_2(3) = \min \{ 0 + 1; 1 + \infty; 9 + \infty; 7 + \infty; 2 + \infty \} = 1.$$

$$\lambda_3(3) = \min \{ 0 + \infty; 1 + 8; 9 + \infty; 7 + 2; 2 + \infty \} = 9.$$

$$\lambda_4(3) = \min \{ 0 + \infty; 1 + 7; 9 + 1; 7 + \infty; 2 + 4 \} = 6.$$

$$\lambda_5(3) = \min \{ 0 + 3; 1 + 1; 9 - 5; 7 + \infty; 2 + \infty \} = 2.$$

Отримані значення  $\lambda_i(3)$  занесемо в четвертий стовпець таблиці. Величини  $\lambda_i(3)$  дорівнюють довжині мінімального шляху з першої вершини в  $i$ -ту, що містить не більш трьох дуг.

$$k = 4. \lambda_1(4) = 0.$$

Рівність (4.5) для  $k = 4$  має вигляд:

$$\lambda_i(4) = \min_{1 \leq j \leq 5} \{ \lambda_j(3) + c_{ji} \}$$

$$\lambda_2(4) = \min \{ 0 + 1; 1 + \infty; 9 + \infty; 6 + \infty; 2 + \infty \} = 1.$$

$$\lambda_3(4) = \min \{0 + \infty; 1 + 8; 9 + \infty; 6 + 2; 2 + \infty\} = 8.$$

$$\lambda_4(4) = \min \{0 + \infty; 1 + 7; 9 + 1; 6 + \infty; 2 + 4\} = 6.$$

$$\lambda_5(4) = \min \{0 + 3; 1 + 1; 9 - 5; 6 + \infty; 2 + \infty\} = 2$$

Отримані значення  $\lambda_i(4)$  занесемо в п'ятий стовпець таблиці. Величини  $\lambda_i(4)$  дорівнюють довжині мінімального шляху з першої вершини в  $i$ -у, що містить не більше чотирьох дуг.

Таблиця 3.1

$i$ (номер вершини)	$\lambda_i(0)$	$\lambda_i(1)$	$\lambda_i(2)$	$\lambda_i(3)$	$\lambda_i(4)$
1	0	0	0	0	0
2	$\infty$	1	1	1	1
3	$\infty$	$\infty$	9	9	8
4	$\infty$	$\infty$	7	6	6
5	$\infty$	3	2	2	2

*Крок 5.* Відновлення мінімального шляху.

Для останньої вершини  $v_3$  попередню їй вершину  $v_r$  визначають зі співвідношення (3.6), отриманого при  $s = 3$ :

$$\lambda_r(3) + c_{r3} = \lambda_3(4), v_r \in G^{-1}(v_3), \quad (3.7)$$

де  $G^{-1}(v_3)$  – прообраз вершини  $v_3$ .

$$G^{-1}(v_3) = \{v_2, v_4\}.$$

Підставимо в (3.7) послідовно  $r = 2$  та  $r = 4$ , щоб визначити, для якого  $r$  ця рівність виконується:

$$\lambda_2(3) + c_{23} = 1 + 8 \neq \lambda_3(4) = 8,$$

$$\lambda_4(3) + c_{43} = 6 + 2 = \lambda_3(4) = 8,$$

Таким чином, вершиною, що передуює вершині  $v_3$ , є вершина  $v_4$ .

Для вершини  $v_4$  попередня їй вершина  $v_r$  визначається зі співвідношення (3.6), отриманого при  $s = 4$ :

$$\lambda_r(2) + c_{r4} = \lambda_4(3), v_r \in G^{-1}(v_4), \quad (3.8)$$

де  $G^{-1}(v_4)$  – прообраз вершини  $v_4$ .

$$G^{-1}(v_4) = \{v_2, v_3, v_5\}.$$

Підставимо в (3.8) послідовно  $r = 2$ ,  $r = 3$  й  $r = 5$ , щоб визначити, для якого

$r$  ця рівність виконується:

$$\lambda_2(2) + c_{24} = 1 + 7 \neq \lambda_4(3) = 6,$$

$$\lambda_3(2) + c_{34} = 1 + 1 \neq \lambda_4(3) = 6,$$

$$\lambda_5(2) + c_{54} = 2 + 4 = \lambda_4(3) = 6$$

Таким чином, вершиною, що передуює вершині  $v_4$ , є вершина  $v_5$ .

Для вершини  $v_5$  попередня їй вершина  $v_r$  визначається зі співвідношення (3.6), отриманого при  $s = 5$ :

$$\lambda_r(1) + c_{r5} = \lambda_5(2), v_r \in G^{-1}(v_5), \quad (3.9)$$

де  $G^{-1}(v_5)$  – прообраз вершини  $v_5$ .

$$G^{-1}(v_5) = \{v_1, v_2\}.$$

Підставимо в (3.9) послідовно  $r = 1$  й  $r = 2$ , щоб визначити, для якого  $r$  ця рівність виконується:

$$\lambda_1(1) + c_{15} = 0 + 3 \neq \lambda_5(2) = 2,$$

$$\lambda_2(1) + c_{25} = 1 + 1 = \lambda_5(2) = 2.$$

Таким чином, вершиною, що передуює вершині  $v_5$ , є вершина  $v_2$ .

Для вершини  $v_2$  попередня їй вершина  $v_r$  визначається зі співвідношення (3.6), отриманого при  $s = 2$ .

$$\lambda_r(0) + c_{r2} = \lambda_2(1), v_r \in G^{-1}(v_2), \quad (3.10)$$

де  $G^{-1}(v_2)$  – прообраз вершини  $v_2$ .

$$G^{-1}(v_2) = \{v_1\}.$$

Підставимо в (4.10)  $r = 1$ , щоб визначити, чи виконується ця рівність:

$$\lambda_1(0) + c_{12} = 0 + 1 = \lambda_2(1) = 1$$

Таким чином, вершиною, що передуює вершині  $v_2$ , є вершина  $v_1$ .

Отже, знайдений мінімальний шлях  $-v_1, v_2, v_5, v_4, v_3$ , його довжина дорівнює 8.

### Опис машинного алгоритму Форда-Беллмана

На відміну від алгоритму Дейкстри, цей алгоритм застосовний також і до графів, що містять ребра з від'ємною вагою. Втім, якщо граф містить від'ємний цикл, то, зрозуміло, найкоротшого шляху до деяких вершин може не існувати (у зв'язку з тим, що вага найкоротшого шляху має дорівнювати мінус нескінченності); втім, цей алгоритм можна модифікувати, щоб він сигналізував про наявність циклу з від'ємною вагою, або навіть виводив сам цей цикл.

Спочатку будемо вважати, що граф не містить циклу з від'ємною вагою. Створимо список відстаней  $d[0 \dots n-1]$ , який після відпрацювання алгоритму буде містити відповідь. На початку роботи ми заповнюємо його наступним чином:  $d[v] = 0$ , а всі інші елементи  $d[]$  дорівнюють нескінченності ( $\inf$ ).

Алгоритм Форда-Беллмана складається з кількох фаз.

На кожній фазі проглядають всі ребра графа, і алгоритм намагається зробити релаксацію (relax, ослаблення) уздовж кожного ребра  $(a, b)$  з вагою  $c$ .

Релаксація уздовж ребра – це спроба поліпшити значення  $d[b]$  значенням  $d[a] + c$ . Фактично це означає, що ми намагаємося поліпшити відповідь для вершини  $b$ , користуючись ребром  $(a, b)$  і поточною відповіддю для вершини  $a$ .

Стверджується, що достатньо  $(n-1)$ -ї фази алгоритму, щоб коректно порахувати довжини всіх найкоротших шляхів в графі. Для недосяжних вершин відстань  $d[]$  залишиться рівною  $inf$ .

Для алгоритму Форда-Беллмана, на відміну від багатьох інших алгоритмів на графах, зручніше представляти граф у вигляді одного списку всіх ребер (а не  $n$  списків ребер, що інцидентні з кожною з вершин). У наведеній реалізації використаємо структуру даних `edge` для ребра. Вхідними даними для алгоритму є числа  $n, m$ , список  $e$  ребер, і номер стартової вершини  $v$ . Всі номери вершин нумеруються з 0 по  $n-1$ .

Константа `inf` позначає число "нескінченність".

### Псевдокод

```
прочитати e // e[0 ... m - 1] - список, в якому
зберігаються ребра та їх ваги (first, second - вершини,
що з'єднані ребром, value - вага ребра)
for i = 0 ... n - 1
    d[i] = inf
d[0] = 0
for i = 1 ... n
    for j = 0 ... m - 1
        if d[e[j].second] > d[e[j].first] + e[j].value
            d[e[j].second] = d[e[j].first] + e[j].value
        if d[e[j].first] > d[e[j].second] + e[j].value
            d[e[j].first] = d[e[j].second] + e[j].value
вивести d
```

### 3.6.3. АЛГОРИТМ ФЛОЙДА-УОРШЕЛЛА

Даний алгоритм іноді називають алгоритмом Флойда-Уоршелла. Алгоритм Флойда-Уоршелла є алгоритмом на графах, який розроблений в 1962 році Робертом Флойдом і Стивеном Уоршеллом. Він служить для знаходження найкоротших шляхів між усіма парами вершин графа.

Метод Флойда безпосередньо ґрунтується на тому факті, що в графі з додатними вагами ребер будь-який неелементарний (більше 1 ребра) найкоротший шлях складається з інших найкоротших шляхів.

Цей алгоритм більш загальний у порівнянні з алгоритмом Дейкстри, тому що він знаходить найкоротші шляхи між будь-якими двома вершинами графа.

В алгоритмі Флойда використовують матрицю  $A$  розмірності  $n \times n$ , у якій обчислюють довжини найкоротших шляхів. Елемент  $A[i, j]$  дорівнює відстані від вершини  $i$  до вершини  $j$ , яка має кінцеве значення, якщо існує ребро  $(i, j)$ , і дорівнює нескінченності в протилежному випадку.



## Алгоритм Флойда

Основна ідея алгоритму. Нехай є три вершини  $i, j, k$  і задані відстані між ними. Якщо виконується нерівність  $A[i, k] + A[k, j] < A[i, j]$ , то доцільно замінити шлях  $i \rightarrow j$  шляхом  $i \rightarrow k \rightarrow j$ . Таку заміну виконують систематично в процесі виконання даного алгоритму.

*Крок 0.* Визначаємо початкову матрицю відстані  $A_0$  й матрицю послідовності вершин  $S_0$ . Кожний діагональний елемент обох матриць дорівнює 0, таким чином показуючи, що ці елементи в обчисленнях не беруть участь. Встановимо  $k = 1$ .

Основний крок  $k$ . Задаємо рядок  $k$  і стовпець  $k$  як провідний рядок і провідний стовпець. Розглядаємо можливість застосування описаної вище заміни до всіх елементів  $A[i, j]$  матриці  $A_{k-1}$ . Якщо виконується нерівність

$A[i, k] + A[k, j] < A[i, j]$ , ( $i \neq k, j \neq k, i \neq j$ ), тоді виконуємо наступні дії:

1. створюємо матрицю  $A_k$  шляхом заміни в матриці  $A_{k-1}$  елемента  $A[i, j]$  сумою  $A[i, k] + A[k, j]$ ;
2. створюємо матрицю  $S_k$  шляхом заміни в матриці  $S_{k-1}$  елемента  $S[j, j]$  на  $k$ . Встановимо  $k = k + 1$  й повторюємо крок  $k$ .

## Опис машинного алгоритму Флойда-Уоршелла

Дано орієнтований або неорієнтований зважений граф  $G$  з  $n$  вершинами. Потрібно знайти значення всіх величин  $d[i][j]$  – довжини найкоротшого шляху з вершини  $i$  в вершину  $j$ .

Передбачається, що граф не містить циклів від'ємної ваги (тоді відповіді між деякими парами вершин може просто не існувати – він буде нескінченно маленьким).

У списку  $d[0 \dots n-1][0 \dots n-1]$  на  $i$ -й ітерації будемо зберігати відповідь на початкове завдання з обмеженням на те, що як «пересадочні» на нашому шляху будемо використовувати вершини з номером строго менше  $i-1$  (вершини нумеруємо з нуля). Нехай йде  $i$ -та ітерація, і ми хочемо оновити масив до  $i+1$ -ї. Для цього для кожної пари вершин просто спробуємо взяти як «пересадочну»  $i-1$ -у вершину, і якщо це покращує відповідь, то так і залишимо. Всього зробимо  $n+1$  ітерацію, після її завершення як «пересадочну» зможемо використовувати будь-яку, і список  $d$  буде розв'язком, тобто міститиме довжини найкоротшого шляху з вершини  $i$  в вершину  $j$ .

Всього використовуємо  $n$  ітерацій по  $n$  ітерацій по  $n$  ітерацій, разом  $O(n^3)$  операцій.

#### Псевдокод

```

прочитати g // g[0...n-1][0...n-1] - список, що містить
ваги ребер // g[i][j] = inf, якщо ребро між i та j
відсутнє
d = g
for i = 1 ... n + 1
    for j = 0 ... n - 1
        for k = 0 ... n - 1
            if d[j][k] > d[j][i - 1] + d[i - 1][k]
                d[j][k] = d[j][i - 1] + d[i - 1][k]
вивести d

```

### 3.6.4. МЕТОД ДИНАМІЧНОГО ПРОГРАМУВАННЯ

#### (топологічне сортування)

**Пряма ітерація.** Нехай вузли орієнтованого графа пронумеровані так, що дуга  $(x_i, x_j) \in E$  завжди орієнтована від вузла  $x_i$  до вузла  $x_j$ , що має більший номер. Для ациклічного графа така нумерація завжди можлива і залежить від способу задавання графа.

Ациклічними графами будемо називати такі орграфи, що не містять циклів.

Приклад орграфа, який містить цикл, показано на рисунку.

Алгоритм має передбачати перевірку графа, заданого матрицею ваг  $C$ , на ациклічність.

Така перевірка може бути реалізована методом топологічного сортування, або довільним іншим методом.

Початкова вершина  $s$  отримує номер 1, а кінцева  $t$  – номер  $n$ .

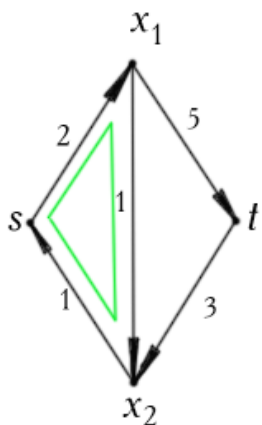
Нехай  $\lambda(x_i)$  – позначка вершини  $x_i$ , яка дорівнює довжині найкоротшого шляху від 1 до  $x_i$ ,  $s$  – початкова вершина (джерело),  $t$  – кінцева вершина (стік).

**Крок 1.** Встановимо  $\lambda(s) = 0$ ,  $\lambda(x_i) = \infty$  для всіх вершин  $x_i \in X \setminus s$ ,  $j = 1$ .

**Крок 2.**  $j := j + 1$ . Присвоїмо вершині  $x_j$  позначку  $\lambda(x_j)$ , яка дорівнює довжині найкоротшого шляху від 1 до  $x_j$ , використовуючи для цього співвідношення

$$\lambda(x_j) = \min_{x_i \in \Gamma^{-1}(x_j)} [\lambda(x_i) + c_{ij}] \quad (3.11)$$

Пояснимо застосування формули (4.11). Для отримання позначки  $\lambda(x_j)$  для вузла  $x_j$  потрібно переглянути всі дуги, що заходять у цю вершину. Внаслідок



такого перегляду вибрати той суміжний вузол  $x_i$ , який має найменше значення позначки  $\lambda(x_i)$

*Крок 3.* Повторювати *крок 2*, поки остання вершина  $n$  не отримає позначку  $\lambda(t)$ .

**Додаткове пояснення.** При формуванні позначки  $\lambda(x_j)$  для вузла  $x_j$  потрібно розуміти, що всі позначки  $\lambda(x_i)$  вершин  $x_i \in \Gamma^{-1}(x_j)$  відомі, тому що у відповідності зі способом нумерації це означає, що  $x_i < x_j$ , а отже, вершини  $x_i$  вже позначені в процесі застосування алгоритму.

Позначка  $\lambda(t)$  дорівнює довжині найкоротшого шляху від  $s$  до  $t$ .

### Зворотна ітерація

При виконанні прямої ітерації знайдено довжину найкоротшого шляху, який дорівнює  $\lambda(t)$ . Зворотна ітерація має на меті знайти сам шлях, який буде представлений послідовністю вершин.

Цей шлях може бути знайдений способом послідовного повернення.

Для знаходження попередньої вершини будемо використовувати вираз (3.11) у вигляді

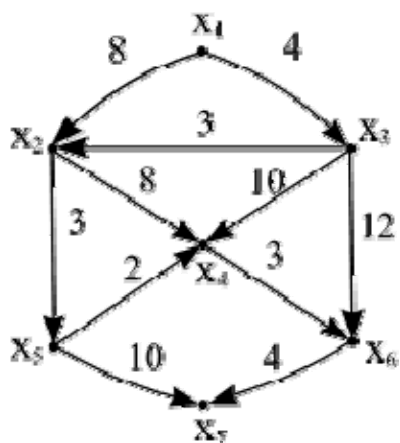
$$\lambda(x_i) = \min_{x_j \in \Gamma^{-1}(x_i)} [\lambda(x_j) - c_{ij}] \quad (3.12)$$

Починаючи з вершини  $t$ , що має номер  $n$ , на кожному кроці, знаючи вершину  $x_j$ , знаходимо вершину  $x_i$ , для якої виконується співвідношення (3.12), і так продовжуємо доти, поки не буде досягнута початкова вершина (тобто поки не буде  $x_i = s$ ).

Тоді найкоротший шлях буде представлений послідовністю вершин:

$$s \rightarrow \dots \rightarrow x_i \rightarrow x_j \rightarrow \dots \rightarrow t$$

### 3.6.5. КОНТРОЛЬНИЙ ПРИКЛАД



Для графа, зображеного на рис. 3.11, визначимо найкоротший шлях між вершинами  $x_1$  і  $x_7$ , використовуючи метод динамічного програмування.

Оскільки граф містить дуги  $(x_3, x_2)$  і  $(x_5, x_4)$ , що мають неправильну нумерацію (від більшого до меншого), необхідно перенумерувати вершини графа, застосовуючи алгоритм топологічного сортування

Рис. 3.11. вершин.

У нашому випадку з графа будуть послідовно виключатися вершини  $x_7, x_6, x_4, x_5, x_2, x_3, x_1$ . Відповідно, вершини графа отримають нову нумерацію, і граф буде мати вигляд, представлений на рис. 3.12 (стара нумерація вершин збережена в дужках).

На першому кроці оцінка  $\lambda(x_1)=0$ . Переходимо до вершини  $x_2$ .

Множина  $\Gamma^{-1}(x_2)$  включає тільки одну вершину  $x_1$ . Отже, оцінка для вершини  $x_2$ , яка визначається за формулою (3.11), буде  $\lambda(x_2)=\min\{0+4\}=4$ . Переходимо до вершини  $x_3$ . Для вершини  $x_3$  множина  $\Gamma^{-1}(x_3)=\{x_1, x_2\}$ . У цьому випадку оцінка буде вибиратися як мінімальна з двох можливих:  $\lambda(x_3)=\min\{0+8, 4+3\}=7$ . Для вершини  $x_4$  оцінка визначається знову однозначно:  $\lambda(x_4)=\min\{7+3\}=10$ . Однак при переході до вершини  $x_5$  ми отримуємо відразу три вхідні дуги  $(x_2, x_5), (x_3, x_5), (x_4, x_5)$ . Застосовуючи формулу (3.11), визначаємо оцінку для вершини  $x_5$ :  $\lambda(x_5)=\min\{4+10, 7+8, 10+2\}=12$ .

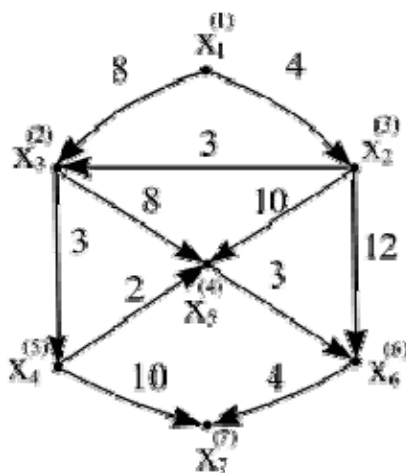


Рис. 3.12.

Далі аналогічним чином вершина  $x_6$  отримує оцінку  $\lambda(x_6)=\min\{4+12, 12+3\}=15$  і, нарешті, вершина  $x_7$  отримує оцінку  $\lambda(x_7)=\min\{10+10, 15+4\}=19$ .

Таким чином, кінцева вершина  $x_7$  шляху  $\mu(x_1, x_7)$  досягнута, довжина шляху дорівнює  $L(\mu)=19$ .

Застосовуючи вираз (3.12), послідовно визначаємо вершини, які входять в найкоротший шлях.

Переміщуючись від кінцевої вершини  $x_7$ , вибираємо послідовність вершин, для якої вираз (3.12) набуває значення «істинно»:  $x_6, x_5, x_4, x_3, x_2, x_1$ , тобто

найкоротший шлях проходить послідовно через усі вершини графа.

Дійсно, легко переконатися в істинності виразів

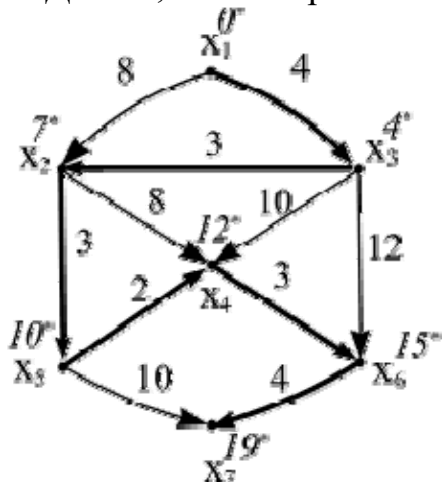


Рис. 3.13.

$$\lambda(x_7) = \lambda(x_6) + c_{67}, \quad (19 = 15 + 4);$$

$$\lambda(x_6) = \lambda(x_5) + c_{56}, \quad (15 = 12 + 3);$$

$$\lambda(x_5) = \lambda(x_4) + c_{45}, \quad (12 = 10 + 2);$$

$$\lambda(x_4) = \lambda(x_3) + c_{34}, \quad (10 = 7 + 3);$$

$$\lambda(x_3) = \lambda(x_2) + c_{23}, \quad (7 = 4 + 3);$$

$$\lambda(x_2) = \lambda(x_1) + c_{12}, \quad (4 = 0 + 4);$$

Зробивши перенумерацію вершин графа на початкову, остаточно визначаємо найкоротший

шлях:  $\mu(x_1, x_7) = \{x_1, x_3, x_2, x_5, x_4, x_6, x_7\}$ .

Задачу вирішено. Розв'язок у вигляді виділеного шляху зображений на рис. 3.13. Курсивом «із зірочкою» виділені значення позначок вершин  $\lambda(x_i)$ .

### Опис машинного алгоритму топологічного сортування

Рекурсивний варіант процедури пошуку вглиб дозволяє досить просто побудувати топологічне сортування у ациклічному графі. В наведеному нижче псевдокоді представлені дві процедури: `TopologicalSort(G)`, `DFSR(G, s)`. Для кожної вершини  $v$  вводиться параметр  $f[v]$  – номер вершини у топологічному сортуванні. У даному псевдокоді алгоритму змінна `current_label` містить поточний номер у топологічному сортуванні, який буде приписано черговій розглянутій вершині.

**TopologicalSort**(граф  $G$ )

1. позначити всі вершини як не відвідані
2. `current_label`  $\leftarrow n$  (кількість вершин графу)
3. для кожної вершини  $v$  графу  $G$ :
4. if вершина  $v$  ще не відвідана
5. `DFSR(G, v)`

**DFSR**(граф  $G$ , початкова вершина  $s$ )

1. позначити  $s$  як відвідану
2. для кожного ребра  $(s, u)$  в  $G$ :
3. if вершина  $u$  ще не відвідана
4. `DFSR(G, u)`
5. `f[s]  $\leftarrow$  current_label`
6. `current_label`  $\leftarrow$  `current_label` - 1

### 3.6.6. АЛГОРИТМ ПОШУКУ ШЛЯХІВ У ШИРИНУ

Алгоритм пошуку в ширину (англ. Breadth-first search, BFS) дозволяє знайти найкоротші шляхи з однієї вершини незваженого (орієнтованого або неорієнтованого) графа до всіх інших вершин. Під найкоротшим шляхом розуміємо шлях, що містить найменше число ребер.

Алгоритм побудований на простій ідеї – нехай до деякої вершини  $u$  знайдено найкоротшу відстань і вона дорівнює  $d$ , а до вершини  $v$  найкоротша відстань не менша, ніж  $d$ . Тоді якщо вершини  $u$  і  $v$  – суміжні, то найкоротша відстань до вершини  $v$  дорівнює  $d + 1$ .

Через  $d[i]$  будемо позначати найкоротшу відстань до вершини  $i$ . Нехай початкова вершина має номер  $s$ , тоді  $d[s] = 0$ . Для всіх вершин, суміжних з  $s$ , відстань дорівнює 1, для вершин, суміжних з тими, до яких відстань дорівнюватиме 1, відстань до початкової вершини дорівнюватиме 2 (якщо тільки вона не дорівнює 0 або 1) і т. д.

Отже, організувати процес обчислення найкоротших відстаней до вершин можна наступним чином. Для кожної вершини в списку  $d$  будемо зберігати найкоротшу відстань до цієї вершини, якщо ж відстань невідома – будемо зберігати значення  $-1$  або `None` (в мові Python). На початку відстань до всіх вершин дорівнює  $-1$  (`None`), крім початкової вершини, до якої відстань дорівнює  $0$ . Потім перебираємо всі вершини, до яких відстань дорівнює  $0$ , перебираємо суміжні з ними вершини і для них записуємо відстань  $1$ . Потім перебираємо всі вершини, до яких відстань дорівнюватиме  $1$ , перебираємо їх сусідів, записуємо для них відстань  $2$  (якщо вона до цього була  $-1$  (`None`)). Потім перебираємо вершини, до яких відстань дорівнювала  $2$  і тим самим визначаємо вершини, до яких відстань дорівнюватиме  $3$  і т. д. Цей цикл можна повторювати або поки виявляються нові вершини на черговому кроці, або  $n-1$  раз (де  $n$  – число вершин в графі), оскільки довжина найкоротшого шляху в графі не може перевищувати  $n-1$ .

Така реалізація алгоритму буде неефективною, якщо на кожному кроці перебирати всі вершини, відбираючи ті, які були виявлені на останньому кроці. Для ефективної реалізації слід використовувати чергу.

У чергу потрібно «закладати» вершини **після того**, як до них буде **визначена** найкоротша відстань. Тобто черга міститиме вершини, які були «виявлені» алгоритмом, але не були розглянуті ребра, що виходять з цих вершин. Можна також сказати, що це черга на «обробку» вершин.

З черги послідовно беруть вершини, розглядають всі «вихідні» (ті, що з них виходять) для них ребра. Якщо ребро веде в невиявлену до цього вершину, тобто відстань до нової вершини не визначена, то її встановлюють на одиницю більшою, ніж відстань до оброблюваної вершини, а нову вершину додають в кінець черги.

Таким чином, якщо з черги дістаємо вершину з відстанню  $d$ , то в кінець черги будуть додаватися вершини з відстанню  $d + 1$ , тобто в будь-який момент виконання алгоритму черга складається з вершин, віддалених на відстань  $d$ , за якими слідує вершини, віддалені на відстань  $d + 1$ .

### **Опис машинного алгоритму пошуку в ширину**

Запишемо алгоритм пошуку в ширину.

```
D = [None] * (n + 1)
D[start] = 0
Q = [start]
Qstart = 0
while Qstart < len(Q):
    u = Q[Qstart]
    Qstart += 1
    for v in V[u]:
        if D[v] is None:
            D[v] = D[u] + 1
            Q.append(v)
```

У цьому алгоритмі  $n$  – число вершин у графі, пронумерованих від 1 до  $n$ . Номер початкової вершини (від якої шукають шляхи) зберігають в змінній `start`.  $Q$  – черга, в якій зберігають оброблювані елементи. Для цього використаємо список. `Qstart` – перший елемент черги, додавання нової вершини в кінець черги – це виклик методу `append` для списку, видалення вершини з початку черги – це збільшення `Qstart` на 1 (при цьому перший елемент в черзі зберігається в `Q[Qstart]`).

На початку в чергу додають тільки один елемент `start`, для якого на самому початку визначено відстань `D[start]=0` (для всіх інших елементів відстань не визначена).

Цикл триває, поки черга не порожня, що перевіряють умовою `Qstart < len(Q)`. У циклі з черги видаляють перший елемент `u`. Потім перебирають всі суміжні з ним вершини `v`.

Якщо вершина `v` була виявлена раніше (що перевіряють за допомогою умови `D[v]==-1` або `D[v] is None`), то відстань до вершини `v` встановлюють такою, що дорівнює відстані до вершини `u`, збільшеній на 1, потім вершину `v` додають в кінець черги.

Якщо граф має  $n$  вершин і  $m$  ребер, то складність такого алгоритму дорівнює  $O(n+m)$ .

Алгоритму необхідно пройти по всіх ребрах. Тому якщо граф зберігають за допомогою матриці суміжності, складність алгоритму дорівнює  $O(n^2)$ . Внутрішній цикл перебору всіх суміжних вершин буде містити  $n$  кроків для кожної обробленої вершини графа.

Приклад програми за алгоритмом пошуку в ширину:

```
def BFS(s, Adj):
    level = { s: 0 }
    parent = { s: None }
    i = 1
    frontier = [s]
    while frontier:
        next = []
        for u in frontier:
            for v in Adj[u]:
                if v not in level:
                    level[v] = i
                    parent[v] = u
                    next.append(v)
        frontier = next
        i += 1
```

### 3.6.7. МЕТОД ПОШУКУ ШЛЯХІВ У ГЛИБИНУ

Обхід у глибину (англ. Depth-First Search, DFS) – один з основних рекурсивних методів обходу вершин графа. Загальна ідея алгоритму полягає в наступному: для кожної не відвіданої вершини необхідно знайти всі не відвідані суміжні вершини і повторити пошук для них. Таким чином, стратегія обходу в глибину наступна: йти «вглиб», поки це можливо, коли є не пройдені «вихідні» (ті, що виходять з вершини) ребра, і повертатися, щоб шукати інший шлях, коли таких ребер немає. Вперше «відвідавши» вершину  $v$  графа, відзначаємо цю подію, проставляючи в масиві `used` пройдених вершин значення `true`, `used[v] ← true`. В результаті обходу в глибину знаходяться перші лексикографічні шляхи в графі з відвіданої вершини в усі досяжні з неї.

Початковий граф може бути як неорієнтованим, так і орієнтованим, суті методу це не змінює.

#### Покрокове представлення

$G(V, E)$  – початковий граф, в якому  $V$  – це множина вершин, а  $E$  – це множина ребер.

`used` – масив для зберігання інформації про пройдені і не пройдені вершини графа  $G$ .

```
for (для) всіх вершин  $v \in V[G]$  # помічаємо кожну вершину як не відвідану  
used[v] ← false
```

```
DFS(v)  
    if used[v]=true  
        return  
    used[v] ← true # відмічаємо вершину, як відвідану  
    for (для) всіх вершин  $w \in E[v]$  # обробляємо ребро  $(v, w)$   
        DFS(w) # викликаємо функцію рекурсивно з кожної  
        вершини  $w$ , яка суміжна з вершиною  $v$ 
```

#### Класифікація ребер при обході в глибину

Дуги орієнтованого графа поділяють на категорії залежно від їх ролі в обході в глибину. Цю класифікацію застосовують при вирішенні різних завдань, що базуються на методі обходу. Наприклад, доведено, що орієнтований граф не має циклів тоді і тільки тоді, коли обхід в глибину не знаходить в ньому «зворотних» дуг.

Після роботи обходу в глибину на орієнтованому графі  $G$  побудований ліс  $G_\pi$ .

Дуги в цьому лісі класифікуються як наступні:



1. Дуги дерева – це безпосередньо дуги, з яких складається ліс  $G_\pi$ , побудований в результаті обходу.
2. Зворотні дуги – це все дуги  $(u, v)$ , що з'єднують вершину  $u$  з її предком  $v$  в одному з дерев лісу  $G_\pi$  обходу в глибину. (Зустрічні дуги, що утворюють цикл довжини 2, можливі в орієнтованих графах, вважаються зворотними дугами також).
3. Прямі дуги – це дуги, що з'єднують вершину з її нащадком, але не входять до лісу  $G_\pi$  обходу в глибину.
4. Перехресні дуги – це всі інші дуги орієнтованого графа, які не попали в одну з перерахованих вище категорій. Такі дуги, наприклад, можуть з'єднувати дві вершини одного дерева обходу, якщо жодна з цих вершин не є предком іншої.

Неорієнтовані графи вимагають особливого розгляду, оскільки одне і те ж ребро  $(u, v) = (v, u)$  обробляється двічі, з двох кінців, і може потрапити в різні категорії. Будемо відносити його до тієї категорії, яка представлена раніше в вищенаведеній класифікації. За такої умови прямих і перехресних ребер в неорієнтованому графі не буде.

### Опис машинного алгоритму пошуку в глибину

Заданий неорієнтований граф, що складається з  $N$  вершин і  $M$  ребер, який задають списком ребер. Потрібно зробити обхід в глибину з усіх ще не відвіданих вершин графа за порядком збільшення їх номерів.

З метою здійснення обходу в глибину граф  $G$  зручно представляти в пам'яті комп'ютера списком суміжності  $adj$ , зберігаючи для кожної вершини графа  $v$  список  $adj[v]$  суміжних з нею вершин. Список  $used$ , який виконує роль булевого масиву розмірності  $N$ , служить для відмітки про те, чи стала вершина  $v$  уже відвіданою в процесі обходу в глибину, чи ще ні. При цьому, якщо  $used[v] = \text{True}$ , то вершина  $v$  є відвіданою, а якщо  $used[v] = \text{False}$ , то ні.

1. `adj = [[] for i in range(n)]` *#генератор списку суміжності*

2. `used = [False for i in range(n)]`

*#генератор списку для зберігання інформації про відвідані і не відвідані вершини*

Для представлення графа мовою Python 3.0 зручно використовувати список списків.

У наведеній нижче реалізації дані зчитуються і виводяться в консоль.

#### Вхідні дані

У першому рядку вхідного файлу задано два цілих числа:  $N$  – кількість вершин в графі і  $M$  – кількість ребер графа відповідно. Кожний з наступних  $M$  рядків містить опис ребра графа – два цілих числа з діапазону від 1 до  $N$  – номери початкової та кінцевої вершин.

### Приклад програми

```
n, m = map(int, input().split()) #кількість вершин і
ребер в графі
adj = [[] for i in range(n)] #список суміжності
used = [False for i in range(n)]
#список для інформації про відвідані и не відвідані
вершини

#зчитуємо граф, що заданий списком ребер
for i in range(m):
    v, w = map(int, input().split())
    v -= 1
    w -= 1
    adj[v].append(w)
    adj[w].append(v)

def dfs(v): #процедура обходу в глибину
    if used[v]: #якщо вершина відвідана, то не викликаємо
функцію
        return
    used[v] = True #помічаємо вершину як відвідану
    print(v + 1, end=' ')

    for w in adj[v]:
        dfs(w) #запускаємо обход з усіх вершин, що сміжні
з v

def run():
    for v in range(n):
        dfs(v)
run()
```

### 3.6.8. АЛГОРИТМ ЛЕВІТА

Алгоритм Левіта (англ. Levit's algorithm) знаходить відстань від заданої вершини  $s$  до всіх інших. Дозволяє працювати з ребрами від'ємної ваги за відсутності від'ємних циклів.

Нехай  $d_i$  – поточна довжина найкоротшого шляху до вершини  $i$ . Спочатку, всі елементи  $d$ , крім  $s$ -го, дорівнюють нескінченності;  $d[s] = 0$ .

Розділимо вершини на три множини:

$M_0$  – вершини, відстань до яких вже обчислено (можливо, не до кінця),

$M_1$  – вершини, відстань до яких обчислюється. Ця множина, в свою чергу, ділиться на дві черги:

$M'_1$  – основна черга,

$M''_1$  – термінова черга;

$M_2$  – вершини, відстань до яких ще не обчислено.

Спочатку все вершини, крім  $s$ , поміщають в множину  $M_2$ .

Вершину  $s$  поміщають в множину  $M_1$  (в будь-яку з черг).

**Крок алгоритму:** вибирають вершину  $u$  з  $M_1$ . Якщо черга  $M_1''$  не порожня, то вершина береться з неї, інакше з  $M_1'$ . Для кожного ребра  $(u, v) \in E$  можливі три випадки:

1. Якщо  $v \in M_2$ , то  $v$  переносяться в кінець черги  $M_1'$ . При цьому  $d_v \leftarrow d_u + w_{uv}$  (відбувається релаксація ребра  $uv$ ).

2. Якщо  $v \in M_1$ , то відбувається релаксація ребра  $uv$ .

3. Якщо  $v \in M_0$ , то якщо при цьому  $d_v > d_u + w_{uv}$ , то відбувається релаксація ребра  $uv$  і вершину  $v$  поміщають в  $M_1''$ ; інакше нічого не робимо.

В кінці кроку поміщаємо вершину  $u$  в множину  $M_0$ . Алгоритм закінчує роботу, коли множина  $M_1$  стає порожньою.

Для зберігання вершин використовуємо такі структури даних:

$M_0$  – хеш-таблиця.

$M_1$  – основна та термінова черга.

$M_2$  – хеш-таблиця.

#### Псевдокод

```
for  $u : u \in V$ 
     $d[u] = \infty$ 
     $d[s] = 0$ 
 $M_1'.push(s)$ 
for  $u : u \neq s$  and  $u \in V$ 
     $M_2.add(u)$ 
while  $M_1' \neq \emptyset$  and  $M_1'' \neq \emptyset$ 
     $u = (M_1'' = \emptyset ? M_1'.pop() : M_1''.pop())$ 
    for  $v : uv \in E$ 
        if  $v \in M_2$ 
             $M_1'.push(v)$ 
             $M_2.remove(v)$ 
             $d[v] = \min(d[v], d[u] + w_{uv})$ 
        else if  $v \in M_1$ 
             $d[v] = \min(d[v], d[u] + w_{uv})$ 
        else if  $v \in M_0$  and  $d[v] > d[u] + w_{uv}$ 
             $M_1''.push(v)$ 
```

$$M_0.remove(v)$$

$$d[v] = d[u] + w_{uv}$$

$$M_0.add(u)$$

### 3.7. Вимоги до програмного забезпечення:

1. Лабораторну роботу виконують з використанням скриптової мови програмування Python.
2. Для написання коду застосувати IDE PyCharm 3 Edu.
3. Для написання GUI застосувати бібліотеку tkinter.
4. Необхідно забезпечити ввід даних з клавіатури та з файлу.
5. Остовне дерево або ребра, що входять до найкоротшого шляху, повинні бути відображені розфарбованими відповідно до результатів роботи Вашого алгоритму. При написанні лабораторної роботи обов'язкове використання пакету NetworkX для роботи з графами.

### Зміст звіту:

1. Титульний лист.
2. Тема завдання.
3. Завдання.
4. Блок-схема алгоритму.
5. Роздруківка тексту програми.
6. Контрольний приклад та результати машинного розрахунку.
7. Висновки по роботі.

### Контрольні запитання

1. Назвіть основні способи представлення графів.
2. Покажіть на прикладі пряму і зворотну відповідність для заданої вершини.
3. Чому дорівнює сума степенів усіх вершин неорієнтованого графа?
4. У чому відмінності матричного представлення орієнтованих і неорієнтованих графів?
5. У чому особливості представлення графа матрицею суміжності?
6. У чому особливості представлення графа матрицею інцидентності?
7. Дайте визначення дерева; орієнтованого дерева.
8. Яке дерево називають остовним?
9. Властивості остовних дерев. Теорема Келі.
10. Що називають коренем дерева?
11. Як перетворити неорієнтоване дерево в орієнтоване?
12. Скільки ребер містить остовне дерево графа?
13. Завдання знаходження найкоротшого остова графа.
14. Наведіть практичні приклади знаходження найкоротшого остова графа.
15. Реалізація алгоритму Прима-Краскала для знаходження найкоротшого остова графа.
16. Дайте визначення шляху, маршруту, ланцюга, контура.
17. Який граф називають зваженим?

18. Як визначають довжину шляху графа?
19. Задача знаходження найкоротшого шляху на графі.
20. Реалізація методу динамічного програмування для знаходження найкоротшого шляху на графі.
21. Обмеження застосування методу динамічного програмування для знаходження найкоротшого шляху на графі.
22. Що називають правильною нумерацією вершин графа?
23. Застосування алгоритму топологічного сортування для перенумерації вершин графа.
24. Застосування алгоритму Дейкстри для пошуку найкоротших шляхів у графі.
25. Основні принципи роботи алгоритму Форда-Беллмана для пошуку найкоротших шляхів у графі.
26. Алгоритм Флойда-Уоршелла для пошуку найкоротших шляхів у графі.

### 3.8. Варіанти для виконання лабораторної роботи

Номер варіанта  $I$  визначають як результат операції  $I = NZK \bmod 10 + 1$ , де  $NZK$  – номер залікової книжки. Номер варіанта відповідає номеру пункту завдання до лабораторної роботи.

№	Опис варіанта
1	<p>1. Вивчити способи представлення графів. Написати програму для перетворення графу, заданого матрицею суміжності, в граф, заданий матрицею інцидентності та в словник ребер.</p> <p>2. За правилом, наданим викладачем, сформувати матрицю суміжності.</p> <p>3. Представити початковий граф, заданий матрицею суміжності, та кінцевий граф, заданий матрицею інцидентності у графічній формі.</p>
2	<p>1. Вивчити способи представлення графів. Написати програму для перетворення графу, заданого списком ребер в матрицю інцидентності та в граф, заданий матрицею суміжності.</p> <p>2. За правилом, наданим викладачем, сформувати матрицю інцидентності.</p> <p>3. Представити початковий граф, заданий матрицею інцидентності, та кінцевий граф, заданий матрицею суміжності, у графічній формі.</p>
3	<p>1. Вивчити основні означення та правила формування остовних дерев у графі. Написати програму побудови найкоротшого остовного дерева за допомогою алгоритму Прима-Краскала. Програма повинна вивести список ребер, що входять в найкоротше остовне дерево.</p> <p>2. За правилом, наданим викладачем, сформувати матрицю суміжності.</p> <p>3. Представити граф, заданий матрицею суміжності, у графічній формі та виділити найкоротше остовне дерево, сформоване за алгоритмом Прима-Краскала.</p>
4	<p>1. Розглянути основні принципи пошуку найкоротших шляхів у графі. Написати програму за алгоритмом Дейкстри для пошуку найкоротших шляхів у графі.</p>

№	Опис варіанта
	<p>2. За правилом, наданим викладачем, сформувати матрицю ваг <math>C = [c_{i,j}]</math> графа.</p> <p>3. Представити граф, заданий матрицею ваг <math>C</math>, у графічній формі та виділити найкоротший шлях між заданими викладачем вершинами, сформований за алгоритмом Дейкстри.</p>
5	<p>1. Розглянути принцип побудови алгоритму Форда-Беллмана та написати програму пошуку найкоротших шляхів у графі за цим алгоритмом.</p> <p>2. За правилом, наданим викладачем, сформувати матрицю ваг <math>C = [c_{i,j}]</math> графа.</p> <p>3. Представити граф, заданий матрицею ваг <math>C</math>, у графічній формі та виділити найкоротший шлях між заданими викладачем вершинами, сформований за алгоритмом Форда-Беллмана.</p>
6	<p>1. Розглянути принцип побудови алгоритму Флойда-Уоршелла та написати програму пошуку найкоротших шляхів у графі за цим алгоритмом.</p> <p>2. За правилом, наданим викладачем, сформувати матрицю ваг <math>C = [c_{i,j}]</math> графа.</p> <p>3. Представити граф, заданий матрицею ваг <math>C</math>, у графічній формі та вивести матрицю найкоротших шляхів між вершинами, сформовану за алгоритмом Флойда-Уоршелла.</p>
7	<p>1. Вивчити принципи застосування алгоритму топологічного сортування до пошуку найкоротших шляхів у графі. Написати програму пошуку найкоротших шляхів у графі з застосуванням алгоритму топологічного сортування.</p> <p>2. За правилом, наданим викладачем, сформувати матрицю ваг <math>C = [c_{i,j}]</math> графа.</p> <p>3. Представити граф, заданий матрицею ваг <math>C</math>, у графічній формі та виділити найкоротший шлях між заданими викладачем вершинами, сформований за алгоритмом топологічного сортування.</p>
8	<p>1. Розглянути алгоритм пошуку шляхів у ширину та написати програму пошуку найкоротших шляхів у графі за цим алгоритмом.</p> <p>2. Знайти найкоротший шлях між вершинами, номери яких задані викладачем.</p> <p>3. Представити граф у графічній формі та виділити найкоротший шлях, сформований за алгоритмом пошуку в ширину.</p>
9	<p>1. Розглянути алгоритм пошуку шляхів у глибину та написати програму пошуку найкоротших шляхів у графі за цим алгоритмом.</p> <p>2. Знайти найкоротший шлях між вершинами, номери яких задані викладачем.</p> <p>3. Представити граф у графічній формі та виділити найкоротший шлях, сформований за алгоритмом пошуку в глибину.</p>
10	<p>1. Розглянути алгоритм Левіта пошуку шляхів у графі та написати програму пошуку найкоротших шляхів у графі за цим алгоритмом.</p> <p>2. Знайти найкоротший шлях між вершинами, номери яких задані викладачем.</p> <p>3. Представити граф у графічній формі та виділити найкоротший шлях, сформований за алгоритмом Левіта.</p>

## Лабораторна робота № 4

**Тема:** «Розфарбовування графа, алгоритми розфарбування».

**Мета роботи:** вивчення способів правильного розфарбовування графа.

**Завдання:** створити програму для правильного розфарбовування графа на основі одного з алгоритмів розфарбування.

### Теоретичні основи

#### 4.1. ОСНОВНІ ОЗНАЧЕННЯ

Різноманітні завдання, що виникають при плануванні виробництва, складанні графіків огляду, зберіганні та транспортуванні товарів та ін., часто можуть бути представлені як задачі теорії графів, тісно пов'язані з так званим «завданням розфарбовування». Графи, що розглядаються в даній лабораторній роботі, є неорієнтованими і такими, що не мають петель.

Граф  $G$  називають  $r$ -хроматичним, якщо його вершини можуть бути розфарбовані з використанням  $r$  кольорів (фарб) так, що не знайдеться двох суміжних вершин одного кольору. Найменше число  $r$ , таке, що граф  $G$  є  $r$ -хроматичним, називають хроматичним числом графа  $G$  і позначають  $\chi(G)$ . Завдання знаходження хроматичного числа графа називають задачею про розфарбовування (або завданням розфарбовування) графа. Відповідне цьому числу розфарбування вершин розбиває множину вершин графа на  $r$  підмножин, кожна з яких містить вершини одного кольору. Ці множини є незалежними, оскільки в межах однієї множини немає двох суміжних вершин.

Завдання знаходження хроматичного числа довільного графа стало предметом багатьох досліджень в кінці XIX і в XX столітті. З цього питання отримано багато цікавих результатів.

Хроматичне число графа не можна знайти, знаючи тільки кількість вершин і ребер графа. Недостатньо також знати степені кожної вершини, щоб обчислити хроматичне число графа. При відомих величинах  $n$  (кількість вершин),  $m$  (кількість ребер) і  $\deg(x_1), \dots, \deg(x_n)$  (степені вершин графа) можна отримати тільки верхню і нижню оцінки для хроматичного числа графа.

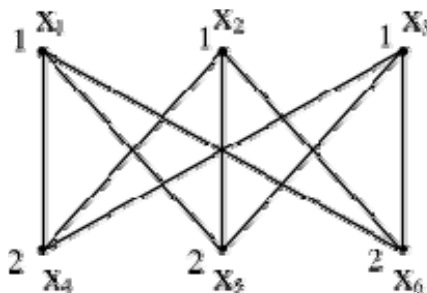


Рис. 4.1 – Дводольний біхроматичний граф Кеніга

Приклад розфарбовування графа наведений на рисунку 4.1. Цей граф є однією із заборонених фігур, що використовуються для визначення планарності. Цифрами «1» і «2» позначені кольори вершин.

Максимальна кількість незалежних вершин графа  $\alpha(G)$ , що дорівнює потужності найбільшої множини попарно несуміжних вершин, збігається також з потужністю найбільшої множини вершин в  $G$ , які можуть бути пофарбовані в один колір, отже:

$$\chi(G) \geq \left\lceil \frac{n}{\alpha(G)} \right\rceil, \quad (4.1)$$

де  $n$  - кількість вершин графа  $G$ , а  $\lceil x \rceil$  позначає найбільше ціле число, яке не більше за  $x$ .

Ще одна нижня оцінка для  $\chi(G)$  може бути отримана наступним чином:

$$\chi(G) \geq \frac{n^2}{n^2 - 2m}. \quad (4.2)$$

Верхня оцінка хроматичного числа може бути обчислена за формулою:

$$\chi(G) \leq 1 + \max_{x_j \in X} [d(x_j) + 1]. \quad (4.3)$$

Застосування оцінок для хроматичного числа значно звужує межі рішення. Для визначення оцінки хроматичного числа також можуть використовуватися інші топологічні характеристики графа, наприклад, властивість планарності.

Граф, який можна зобразити на площині так, що жодні два його ребра не перетинаються між собою, називають *планарним*.

*Теорема про п'ять фарб.* Кожен планарний граф можна розфарбувати за допомогою п'яти кольорів так, що будь-які дві суміжні вершини будуть пофарбовані в різні кольори, тобто якщо граф  $G$  - планарний, то  $\chi(G) \leq 5$ .

*Гіпотеза про чотири фарби (недоведена).* Кожен планарний граф можна розфарбувати за допомогою чотирьох кольорів так, що будь-які дві суміжні вершини будуть пофарбовані в різні кольори, тобто якщо граф  $G$  - планарний, то  $\chi(G) \leq 4$ .

У 1852 р. про гіпотезу чотирьох фарб говорилося в листуванні Огюста де Моргана з сером Вільямом Гамільтоном. З того часу ця «теорема» стала, поряд з теоремою Ферма, однією з найзнаменитіших невирішених задач в математиці.

*Повний граф  $K_n$*  завжди розфарбовується в  $n$  кольорів, тобто кількість кольорів дорівнює кількості його вершин.



## 4.2. АЛГОРИТМ ПРЯМОГО НЕЯВНОГО ПЕРЕБОРУ

Алгоритм прямого неявного перебору є найпростішим алгоритмом вершинного розфарбування графів. Цей алгоритм дозволяє реалізувати правильне розфарбування графа з вибором мінімальної в рамках даного алгоритму кількості фарб.

Введемо такі структури даних:

```
Const Nmax=100; {*максимальна кількість вершин графа*}
Type V=0..Nmax;
      TS=Set of V;
      TColArr = Array (1..Nmax) of V;
      TA = Array (1..Nmax, 1..Nmax) of Integer;

Var ColArr: TColArr; {*масив номерів фарб для кожної вершини графа*}
      A:TA; {*матриця суміжності графа*}

      Function Color (i): Integer;
      {*функція вибору фарби для розфарбування вершини з номером i *}
Var W:TS;
      j:Byte;
Begin
      W:=[];
      For j=1 to i-1 do if A[j,i]=1 then W:=W+[ColArr[j]];
      {*формування множини фарб, що використані для розфарбування суміжних до
      {*вершини i вершин з номерами меншими за i*}
      j:=0; {*змінну j далі використовуємо для вибору номера фарби*}
      Repeat
        Inc(j);
      Until NOT (j In W);
      Color:=j;
End;

Begin
  <Вводимо матрицю суміжності графа>
  {*цикл по вершинах графа*}
  For i=1 to Nmax do ColArr[i]:=Color(i);
  <Виводимо результат розфарбування>
End;
```

### 4.3. ПРИКЛАД АЛГОРИТМУ ПРЯМОГО НЕЯВНОГО ПЕРЕБОРУ

Розглянемо граф  $G(V, E)$ , який показаний на рис. 4.2.

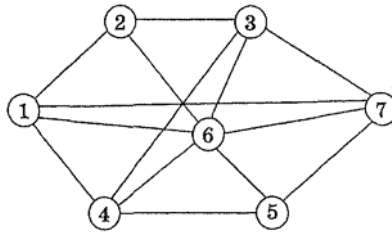


Рис.4.2

Множину вершин графа  $V = \{1, 2, 3, 4, 5, 6, 7\}$  потрібно розфарбувати з використанням алгоритму послідовного розфарбування.

Сформуємо матрицю суміжності  $A$ :

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

**Крок 1.** Для вершини 1, відповідно до представленого вище алгоритму, множина розфарбованих суміжних вершин завжди є пустою. Тому функція  $\text{Color}(1)$  завжди повертатиме фарбу 1. Встановимо, що 1 кодує фарбу червоного кольору.

**Крок 2.** Розглянемо вершину 2. Для цієї вершини єдиною меншою за номером суміжною вершиною є вершина 1. Ця вершина розфарбована червоним кольором. Тому множина  $W$  містить єдиний елемент 1. Тому функція  $\text{Color}(2)$  повертає наступну за номером фарбу 2 синього кольору.

**Крок 3.** Вершина 3 має єдину суміжну вершину з меншим номером. Це вершина 2. Множина  $W$  містить єдиний елемент 2. Тому функція  $\text{Color}(3)$  повертає фарбу з номером 1 червоного кольору.

**Крок 4.** Вершина 4 має дві суміжні вершини з меншими номерами: 1 і 3. Оскільки обидві вершини розфарбовані в колір 1, то множина  $W$  містить єдиний елемент 1. Тому функція  $\text{Color}(4)$  повертає наступну за номером фарбу 2 синього кольору.

**Крок 5.** Вершина 5 має єдину суміжну вершину з меншим номером. Це вершина 4. Множина  $W$  містить єдиний елемент 2. Тому функція  $\text{Color}(5)$  повертає фарбу з номером 1 червоного кольору.

**Крок 6.** Вершина 6 має такі суміжні вершини з меншими номерами: 1, 3, 4 і 5. Ці вершини розфарбовані в колір 1 та колір 2. Отже, множина  $W$  містить два елементи: 1 і 2. Тому функція  $\text{Color}(6)$  повертає наступну за номером фарбу 3 зеленого кольору.

**Крок 7.** Вершина 7 має такі суміжні вершини з меншими номерами: 1, 3, 5 і 6. Ці вершини розфарбовані в колір 1 та колір 3. Отже, множина  $W$  містить два елементи: 1 і 2. Тому функція  $\text{Color}(7)$  повертає фарбу 2 синього кольору.

В результаті роботи данного алгоритму одержуємо правильно розфарбований граф, що показаний на рис. 4.3.

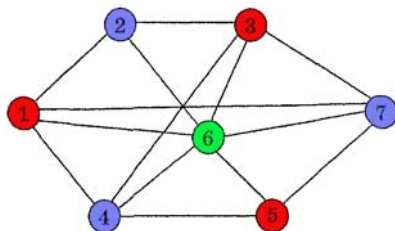


Рис. 4.3.

#### 4.4. ЕВРИСТИЧНИЙ АЛГОРИТМ РОЗФАРБОВУВАННЯ

Точні методи розфарбовування графа складні для програмної реалізації. Однак існує багато евристичних процедур розфарбовування, які дозволяють знаходити хороші наближення для визначення хроматичного числа графа. Такі процедури також можуть з успіхом використовуватися при розфарбовуванні графів з великим числом вершин, де застосування точних методів не виправдане з огляду на високу трудомісткість обчислень.

З евристичних процедур розфарбовування слід зазначити послідовні методи, засновані на впорядкуванні множини вершин. В одному з найпростіших методів вершини спочатку розташовують в порядку зменшення їх степенів. Першу вершину зафарбовують в колір 1, потім список вершин переглядають за зменшенням степенів, і в колір 1 зафарбовують кожну вершину, яка не є суміжною з вершинами, зафарбованими в той же колір. Потім повертаються до першої в списку незафарбованої вершині, фарбують її в колір 2 і знову переглядають список вершин зверху вниз, зафарбовуючи в колір 2 будь-яку незафарбовану вершину, яка не з'єднана ребром з іншою, вже пофарбованою в колір 2, вершиною. Аналогічно діють із кольорами 3, 4 і т. д., доки не будуть пофарбовані всі вершини. Кількість використаних кольорів буде тоді наближеним значенням хроматичного числа графа.

Евристичний алгоритм розфарбовування вершин графа має наступний вигляд:

**Крок 1.** Сортувати вершини графа за степенями зменшення:

$$\deg(x_i) \geq \deg(x_j), \forall x_i, x_j \in G$$

Встановити поточний колір  $p := 1, i := 1$ .

**Крок 2.** Вибрати чергову нерозфарбовану вершину зі списку і призначити їй новий колір  $\text{col}(x_i) := p$ ;  $X = \{x_i\}$ .

**Крок 3.**  $i := i + 1$ . Вибрати чергову не розфарбовану вершину  $x_i$  і перевірити умову суміжності:  $x_i \cap \Gamma(X) = \emptyset$ , де  $X$  - множина вершин, вже розфарбованих в колір  $p$ . Якщо вершина  $x_i$  не є суміжною з даними вершинами, то також присвоїти їй колір  $p$ :  $\text{col}(x_i) := p$  за умови, що вона не має суміжних вершин даного кольору.

**Крок 4.** Повторювати крок 3 до досягнення кінця списку ( $i = n$ ).

**Крок 5.** Якщо всі вершини графа розфарбовані, то – кінець алгоритму; інакше:  $p := p + 1$ ;  $i := 1$ . Повторити крок 2.

Для роботи алгоритму можна використовувати довільну структуру даних, яка однозначно задає граф.

Розглянемо роботу алгоритму на прикладі матриці суміжності  $A$ .

```
import networkx as nx
import warnings
import pylab as plt

A = [[0, 1, 0, 0, 1, 0, 0, 1, 0],
      [1, 0, 1, 0, 0, 0, 1, 0, 0],
      [0, 1, 0, 0, 1, 0, 1, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 1, 0, 0, 0, 0, 1, 0],
      [0, 0, 0, 0, 0, 0, 1, 0, 1],
      [0, 1, 1, 0, 0, 1, 0, 0, 0],
      [1, 0, 0, 0, 1, 0, 0, 0, 1],
      [0, 0, 0, 1, 0, 1, 0, 1, 0]]

Nmax = len(A)
ColArr = [0 for i in range(0, Nmax)]
DegArr = [0 for i in range(0, Nmax)]
SortArr = [i for i in range(0, Nmax)]

print("Нужные для работы массивы длины Nmax")
print("\n", "SortArr: ", SortArr, "\n", "DegArr", DegArr, "\n", "ColArr",
      ColArr, "\n")

print("==Матрица Смежности==")
for i in A:
    print(i)
print("====")

print("Отладка".center(50))
def Degforming():
    for i in range(0, Nmax):
        DegArr[i] = 0
        ColArr[i] = 0
        for j in range(0, Nmax):
            DegArr[i] += A[i][j]
```

```

def Sortnodes():
    for k in range(0, Nmax):
        max_loc = DegArr[k]
        c=k
        for i in range(k + 1, Nmax):
            if DegArr[i] > max_loc: c=i
        DegArr[c], DegArr[k] = DegArr[k], DegArr[c]
        SortArr[c], SortArr[k] = SortArr[k], SortArr[c]

def CheckDop(k): # Функція перевірки кольору суміжних вершин при поточному
розфарбуванні
    p=True
    for j in range(0, Nmax):
        if A[j][k]==1 and ColArr[j]== CurCol: p=False
    return p

def Color(i):
    for j in range(0, Nmax):
        if A[j][i] == 0 and ColArr[j] == 0 and CheckDop(j):
            ColArr[j] = CurCol

CurCol = 1
Degforming()
Sortnodes()
print("SortNodes res:", SortArr)
print("DegArr res:", DegArr)

print("\n\nРозфарбування вершин. На кожній ітерації виводиться масив ColArr")
for n in range(0, Nmax):
    if ColArr[SortArr[n]] == 0:
        ColArr[SortArr[n]] = CurCol
        Color(SortArr[n])
        CurCol += 1
    print(str(ColArr) + "- Ітерація: ", n)

colorized = []
for i in range(0, Nmax): #Исправлена ошибка при формировании данного списка
    colorized.append([i, ColArr[i]])
print("====Результат: [вершина, цвет]====")
print(colorized)
print("\n", "SortArr: ", SortArr, "\n", "DegArr", DegArr, "\n", "ColArr",
ColArr, "\n")

class Graph:
    def __init__(self, matrix, colorized):
        self.color_map = []
        self.graph = nx.Graph()
        self.matrix = matrix
        self.colorized_graph(colorized)
        self.show_graph()

    def colorized_graph(self, colorized):
        for i in range(0, len(colorized)):
            self.color_map.append(colorized[i][1])
            self.graph.add_node(colorized[i][0])

        arcs = []
        for i in range(0, len(self.matrix)):
            for j in range(0, len(self.matrix[i])):
                if i == j or self.matrix[i][j] == 0:
                    continue
                arcs.append((i, j))

```

```

self.graph.add_edges_from(arcs)

def show_graph(self):
    nx.draw(self.graph, node_color=self.color_map, with_labels=True,
font_color='white')
    plt.axis('off')
    plt.show()

```

Graph(A, colorized)

## 4.5. ПРИКЛАД ЕВРИСТИЧНОГО АЛГОРИТМУ РОЗФАРБУВАННЯ

Розфарбуємо граф  $G$ , зображений на рисунку 4.4. Проміжні дані для вирішення завдання будемо записувати в таблицю. Відсортуємо вершини графа за зменшенням їх степенів. У результаті отримуємо вектор відсортованих вершин  $\text{SortArr} = (1, 5, 6, 4, 2, 3)$

Степені, що відповідають даним вершинам, утворюють другий вектор:  $D = (5, 4, 4, 3, 2, 2)$

У першому рядку таблиці запишемо вектор  $\text{SortArr}$ , у другому – степені відповідних вершин. Наступні рядки відображають вміст вектора розфарбування  $\text{ColArr}[\text{SortArr}]$ .

Таким чином, даний граф можна розфарбувати не менш ніж у чотири кольори, тобто  $\chi(G) = 4$ .

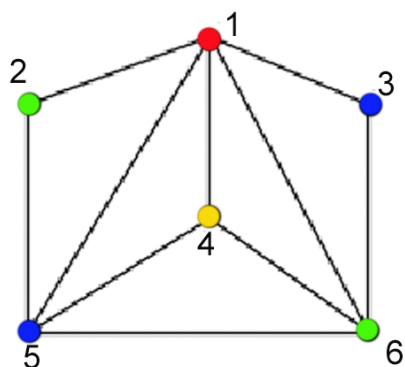


Рис. 4.4.

Номери вершин SortArr	$x_1$	$x_5$	$x_6$	$x_4$	$x_2$	$x_3$
Степені вершин DegArr	5	4	4	3	2	2
CurCol = 1	1	-	-	-	-	-
CurCol = 2	1	2	-	-	-	2
CurCol = 3	1	2	3	-	3	2
CurCol = 4	1	2	3	4	3	2

## 4.6. МОДИФІКОВАНИЙ ЕВРИСТИЧНИЙ АЛГОРИТМ РОЗФАРБУВАННЯ

### Попередні визначення

**Визначення 1.** Відносний степінь – це степінь нерозфарбованих вершин у нерозфарбованому підграфі даного графа.

**Визначення 2.** Двокроковий відносний степінь – сума відносних степенів суміжних вершин у нерозфарбованому підграфі.

Проста модифікація описаної вище евристичної процедури базується на переупорядкуванні нерозфарбованих вершин по незростанню їх відносних степенів.

Дана модифікація полягає у тому, що якщо дві вершини мають однакові степені, то порядок таких вершин випадковий. Їх можна впорядкувати по двокрокових степенях. Двокроковий степінь визначимо як суму відносних степенів суміжних вершин.

**Крок 1.** СОРТУЄМО вершини графа за степенями зменшення:

$$\deg(x_i) \geq \deg(x_j), \forall x_i, x_j \in G$$

У випадку  $\deg(x_i) = \deg(x_j), \forall x_i, x_j \in G$  розглянемо множини суміжності  $\Gamma(x_i)$  і  $\Gamma(x_j)$ .

СОРТУЄМО вершини за ознакою :

$$[\deg(x_{i1}) + \deg(x_{i2}) + \dots + \deg(x_{ik})] \geq [\deg(x_{j1}) + \deg(x_{j2}) + \dots + \deg(x_{jn})],$$

де  $x_{i1}, x_{i2}, \dots, x_{ik}$  - нерозфарбовані вершини з множини суміжності  $\Gamma(x_i)$ ;

$x_{j1}, x_{j2}, \dots, x_{jn}$  - нерозфарбовані вершини з множини суміжності  $\Gamma(x_j)$ ;

Встановити поточний колір  $p := 1, i := 1$ .

**Крок 2.** Вибрати чергову нерозфарбовану вершину зі списку і призначити їй новий колір  $\text{col}(x_i) := p; X = \{x_i\}$ .

**Крок 3.**  $i := i + 1$ . Вибрати чергову нерозфарбовану вершину  $x_i$  і перевірити умову суміжності:  $x_i \cap \Gamma(X) = \emptyset$ , де  $X$  - множина вершин, вже розфарбованих в колір  $p$ . Якщо вершина  $x_i$  не є суміжною з даними вершинами, то також присвоїти їй колір  $p$ :  $\text{col}(x_i) := p$ .

**Крок 4.** Повторювати крок 3 до досягнення кінця списку ( $i = n$ ).

**Крок 5.** Якщо всі вершини графа розфарбовані, то – кінець алгоритму;

інакше:  $p := p + 1; i := 1$ . Повторити крок 2.

Даний алгоритм від попереднього відрізняється ускладненням процедури сортування SortNodes, яка при сортуванні вершин з однаковими степенями враховує двокроковий степінь.

Як і в попередньому випадку, розглянемо роботу алгоритму на прикладі матриці суміжності  $A$ .

**Const** Nmax=100; {\*максимальна кількість вершин графа\*

**Type** TArr = **Array** (1..Nmax) of Integer;

TA = **Array** (1..Nmax, 1..Nmax) of Byte;

**Var** ColArr: TArr; {\*масив номерів фарб для кожної вершини графа\*

```

    DegArr: TArr {*масив степенів вершин*}
    SortArr:TArr;{*відсортований за зменшенням степенів масив
вершин*}
    A:TA; {*матриця суміжності графа*}
    CurCol: Byte; {*поточний номер фарби*}
    n:Byte;

Procedure DegForming;{*Процедура формування масиву степенів
вершин*}
Var k:Byte;
    Function DegCount (m:Byte):Integer;
    Var Deg:Iteger;
    Begin
        Deg:=0;
        For k:=1 to Nmax do Deg:= Deg+A[k,m];
        DegCount:=Deg;
    End;
Begin
    For j:=1 to Nmax do
    begin
        ColArr[i]:=0;
        DegArr[j]:= DegCount(j)*100;
        For i:=1 to Nmax do
            If A[i,j]=1 then DegArr[i]:= DegArr[i]+DegCount(i);
        end;
    End;

Procedure SortNodes; {*Сортування вершин за степенями*}
Var max,c,k,i:Byte;
Begin
    For k:=1 to Nmax-1 do
    begin
        max:=DegArr[k]; c:=k;
        For i:=k+1 to N do
            If DegArr[i] > max then
            begin
                max:= DegArr[ [i];
                c:=i;
            end;
        DegArr[c]:= DegArr[ [k];
        DegArr[k]:=max;
        SortArr[k]:=c;
    end;
    End;
Procedure Color (i:Byte);

```



```

{*Розфарбування поточним кольором не суміжних з i вершин *}
Var j:Byte;
Begin
  For j=1 to Nmax do if A[j,i]=0 then
    begin
      If ColArr[j]=0 then ColArr[j]:=CurCol;
    end;
  End;
Begin
  CurCol:=1;
  <Вводимо матрицю суміжності графа>
  DegForming;  {*Формування масиву степенів вершин*}
  SortNodes;   {*Формування масиву відсортованих вершин SortArr*}
  For n:=1 to Nmax do
    begin
      If ColArr[SortArr[n]]=0 then
        begin
          ColArr[SortArr[n]]:=CurCol;
          Color(SortArr[n]);
          Inc(CurCol);
        end;
    end;
  <Виводимо результат розфарбування>
end;

```

#### 4.7. ПРИКЛАД МОДИФІКОВАНОГО ЕВРИСТИЧНОГО АЛГОРИТМУ РОЗФАРБУВАННЯ

Розфарбуємо граф  $G$ , зображений на рисунку 4.5.

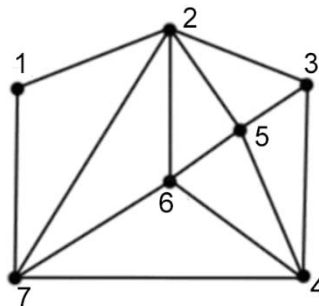


Рис. 4.5.

Відсортуюмо вершини графа за зменшенням їх степенів. У результаті отримуємо вектор відсортованих вершин  $\text{SortArr} = (2, 6, 5, 4, 7, 3, 1)$

Вектор степенів відсортованих вершин має наступний вигляд:  
 $D = (5, 4, 4, 4, 4, 3, 2)$

У першому рядку таблиці запишемо вектор  $\text{SortArr}$ , у другому – вектор  $D$ , а у третьому – вектор  $D^2$ .

Четвертий рядок відповідає представленню степенів  $D$  та  $D^2$  в масиві DegArr.

Наступні рядки відображають вміст вектора розфарбування  $\text{col}(X^*)$ .

Таким чином, даний граф можна розфарбувати не менш ніж у три кольори, тобто  $\chi(G) = 3$ .

Номери вершин $X^*$	$a_2$	$a_6$	$a_5$	$a_4$	$a_7$	$a_3$	$a_1$
Ступінь вершин $D$	5	4	4	4	4	3	2
Двокроковий ступінь $D^2$		17	16	15	15		
DegArr	500	417	416	415	415	300	200
CurCol = 1	1	-	-	1	-	-	-
CurCol = 2	1	2	-	1	-	2	2
CurCol = 3	1	2	3	1	3	2	2

В результаті роботи модифікованого евристичного алгоритму одержимо розфарбований граф, показаний на рис. 4.6.

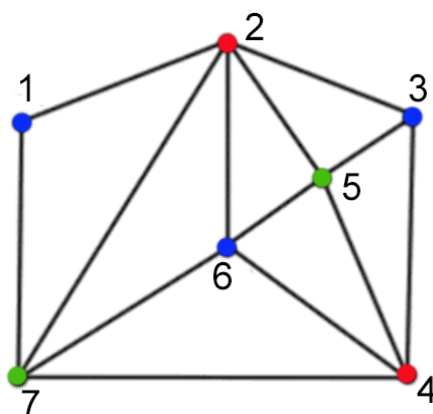


Рис. 4.6.

#### 4.8. РОЗФАРБУВАННЯ ГРАФА МЕТОДОМ А. П. ЄРШОВА

Андрій Петрович Єршов (1931–1988 рр.), видатний вчений в області теоретичного програмування, вніс великий вклад у розвиток інформатики. Зокрема, він створив алгоритм розфарбування графа, що базується на оригінальній евристичній ідеї.

Введемо ряд визначень.

Для даної вершини  $v \in V$  графа  $G(V, E)$  назвемо всі суміжні з нею вершини околom 1-го порядку —  $R_1(v)$ .

Всі вершини, що перебувають на відстані два від  $v$ , назвемо околom 2-го порядку —  $R_2(v)$ .

Граф  $G(V, E)$ , у якого для вершини  $v \in V$  всі інші вершини належать околу  $R_1(v)$ , назвемо граф-зіркою відносно вершини  $v$ .

### Ідея алгоритму

Фарбування у фарбу  $\alpha$  вершини  $v$  утворює навколо неї в  $R_1(v)$  «мертву зону» для фарби  $\alpha$ . Очевидно, при мінімальному розфарбуванні кожна фарба повинна розфарбувати максимально можливу кількість вершин графа. Для цього необхідно, щоб мертві зони, хоча б частково, перекривалися між собою. Перекриття мертвих зон двох несуміжних вершин  $v_1$  і  $v_2$  досягається тільки тоді, коли одна з них перебуває в околі  $R_2(v_1)$  від іншої,  $v_2 \in R_2(v_1)$ .

Таким чином, суть алгоритму полягає в тому, щоб на черговому кроці вибрати для розфарбування фарбою  $\alpha$  вершину  $v_2 \in R_2(v_1)$ . Цей процес повторювати доти, поки фарбою  $\alpha$  не будуть пофарбовані всі можливі вершини графа.

Графічно фарбування вершин  $v_1$  і  $v_2$  однією фарбою можна відобразити як «склеювання» цих вершин.

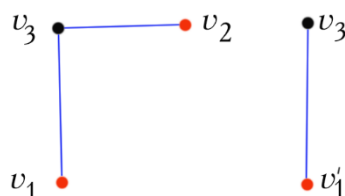


Рис. 4.7. Приклад об'єднання двох вершин:  $v'_1 := v_1 \cup v_2$

При цьому кількість вершин зменшується на одиницю у графі  $G$ , а також зменшується кількість ребер.

### Алгоритм

1. Встановити  $i := 0$ .
  2. Вибрати в графі  $G$  довільну незафарбовану вершину  $v$ .
  3. Встановити  $i := i + 1$ .
  4. Розфарбувати вершину  $v$  фарбою  $i$ .
  5. Розфарбовувати фарбою  $i$  незабарвлені вершини графа  $G$ , вибираючи їх з  $R_2(v)$ , поки граф не перетвориться в граф-зірку відносно  $v$ .
  6. Перевірити, чи залишилися незабарвлені вершини в графі  $G$ . Якщо так, то перейти до п. 2, інакше — до п. 7.
  7. Отриманий повний граф  $K_i$ . Хроматичне число графа  $X(K_i) = i$ .
- Кінець алгоритму.

Як випливає з алгоритму, після того, як у першу обрану вершину стягнуті всі можливі й граф перетворився в граф-зірку відносно цієї вершини, вибирають довільним чином другу вершину й процес повторюють доти, поки граф не перетвориться в повний.

Повний граф – це граф-зірка відносно будь-якої вершини. Хроматичне число повного графа дорівнює числу його вершин.

#### 4.9. ПРИКЛАД РОЗФАРБУВАННЯ ГРАФА МЕТОДОМ А. П. ЄРШОВА

На рис. 4.8 зображений граф  $G$ , на прикладі якого будемо розглядати роботу евристичного алгоритму Єршова.

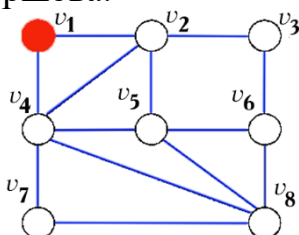


Рис. 4.8. Початковий граф  $G$

Виберемо довільну вершину, наприклад,  $v_1$ . Окіл 2-го порядку  $R_2(v_1) = \{v_3, v_5, v_7, v_8\}$ . Склеїмо вершину  $v_1$  наприклад, з вершиною  $v_3$ :  $v'_1 = v_1 \cup v_3$ . Одержимо граф  $G_1$ , зображений на рис. 4.9.

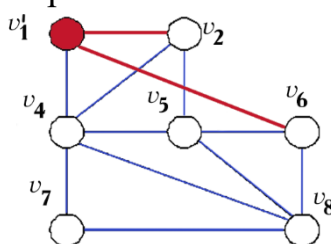


Рис. 4.9. Граф  $G_1$ . Склеєні вершини  $v_1$  та  $v_3$

Розглянемо тепер граф  $G_1$ . Окіл другого порядку вершини  $v'_1$  визначається множиною  $R_2(v'_1) = \{v_5, v_7, v_8\}$ . Склеїмо вершину  $v'_1$ , наприклад, з вершиною  $v_5$ :  $v''_1 := v'_1 \cup v_5$ . Одержимо граф  $G_2$ , зображений на рис. 4.10.

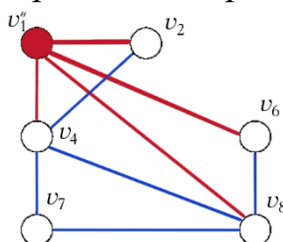


Рис. 4.10. Граф  $G_2$ . Склеєні вершини  $v'_1$  й  $v_5$

У графі  $G_2$  визначимо окіл другого порядку для вершини  $v''_1$ :  $R_2(v''_1) = \{v_7\}$ . Склеїмо вершину  $v''_1$  з вершиною  $v_7$ :  $v'''_1 = v''_1 \cup v_7$ . Одержимо граф  $G_3$ , зображений на рис. 4.11. Цей граф є зіркою відносно вершини  $v'''_1$ .

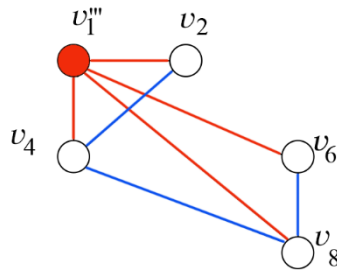


Рис. 4.11. Граф  $G_3$ . Склеєні вершини  $v_1''$  й  $v_7$

У графі  $G_3$  виберемо, наприклад, вершину  $v_2$  для фарбування другою фарбою. Окіл 2-го порядку  $R_2(v_2) = \{v_6, v_8\}$ . Склеїмо вершину  $v_2$ , наприклад, з вершиною  $v_6$ :  $v_1''' = v_1'' \cup v_6$ . Одержимо граф  $G_4$ , зображений на рис. 4.12.

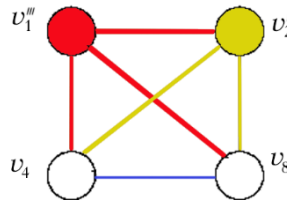


Рис. 4.12. Граф  $G_4$  еквівалентний  $K_4$ . Склеєні вершини  $v_1''$  й  $v_6$

Граф  $G_4$  є повним чотиривершинним графом  $K_4$ . Отже, граф  $G_4$  на рис. 4.12 розфарбовується в чотири кольори. У перший (червоний) колір розфарбовується вершина  $v_1$  й склеєні з нею вершини:  $v_3, v_5$  і  $v_7$ . У другий (жовтий) колір розфарбовується вершина  $v_2$  й склеєна з нею вершина  $v_6$ . У третій (зелений) колір розфарбовується вершина  $v_4$  й у четвертий (білий) колір розфарбовується вершина  $v_8$ .

У підсумку одержуємо правильно розфарбований граф, показаний на рис. 4.13.

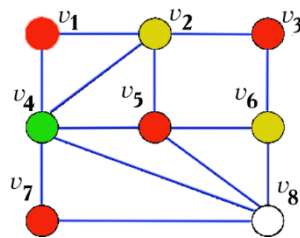


Рис. 4.13. Граф  $G$ , розфарбований за допомогою алгоритму розфарбування А.П.Єршова

### Програмна реалізація

Розглянемо матрицю суміжності графа  $G$ , представленого на рис. 4.8.

$$A = \begin{pmatrix} & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \\ v_1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ v_2 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ v_3 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ v_4 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ v_5 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ v_6 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ v_7 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ v_8 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Операції склеювання відповідних вершин графа, які описані вище, виконують на матриці суміжності шляхом виконання операції диз'юнкції між цими вершинами.

Наприклад, розглянемо склеювання вершин 1 та 3.

Результуюча матриця суміжності містить новий рядок та стовпець для вершини  $v'_1$ , але не містить  $v_1$  та  $v_3$ .

$$A' = \begin{pmatrix} & v'_1 & v_2 & v_4 & v_5 & v_6 & v_7 & v_8 \\ v'_1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ v_2 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ v_4 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ v_5 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ v_6 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ v_7 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ v_8 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Фінальна матриця суміжності має вигляд:

$$A''' = \begin{pmatrix} & v_1''' & v_2' & v_4 & v_8 \\ v_1''' & 0 & 1 & 1 & 1 \\ v_2' & 1 & 0 & 1 & 1 \\ v_4 & 1 & 1 & 0 & 1 \\ v_8 & 1 & 1 & 1 & 0 \end{pmatrix},$$

що відповідає матриці повного графа  $K_4$ .

**Const** n=10; {\*максимальна кількість вершин графа\*}

**Type** V=0..n;

R2S=**Set** of V;

TA = **Array** (1..n, 1..n) of Integer;

**Var** A:TA; {\*матриця суміжності графа\*}

MainNode:Byte;

**Procedure** Glue(master,slave:Byte);

{\*Процедура склеювання вершин\*}

**Begin**

{\*Склеювання рядка master і рядка slave\*}

**For** i:=1 **to** n **do** A[i,master]:= A[i,master] OR A[i,slave];

{\*Склеювання стовпця master і стовпця slave\*}

**For** j:=1 **to** n **do** A[master,j]:= A[master,j] OR A[slave,j];

**End;**

**Procedure** Reduce(master,slave:Byte);

{\*Процедура видалення стовпця і рядка в матриці суміжності\*}

**Begin**

**For** i:=1 **to** n **do**

**For** j:=1 **to** n-1 **do**

{\*Видалення рядка slave\*}

**If** (j≥slave) **then** A[i,j]:= A[i,j+1];

**For** j:=1 **to** n-1 **do**

**For** i:=1 **to** n-1 **do**

{\*Видалення стовпця slave\*}

**If** (i≥slave) **then** A[i,j]:= A[i+1,j];

n:=n-1;

**End;**

**Function** Check\_K:Byte;

{\*Процедура перевірки наявності повного графа\*}

Var Gh:Byte;

**Begin**

{\*У повному графі всі елементи матриці, крім діагональних, повинні бути одиничними\*}

Ch:=0;

**For** i:=1 **to** n **do**

**For** j:=1 **to** n **do if** (i≠j) AND (A[i,j]=0) **then** Ch:=j;

Check\_K:=Ch;

**End;**

**Procrdure** R2(master:Byte);

{\*Процедура формування околу 2-го порядку\*}

**Begin**

**For** j:=1 **to** n **do**

**begin**

**If** (j≠master) AND (A[master,j]=1) **then**

**begin**

{\*Вибрали суміжну вершину до master\*}

**For** i=1 **to** n **do**

```

begin
  If (i≠master) AND (A[j,i]=1) then
    { * Вибрали вершину околу 2-го порядку до master* }
    R2S:=R2S+[i]; { *Додали вершину до множини вершин околу* }
  end;
end;
end;
End;

```

```

Begin
  MainNode:=1; { *Вибираємо першу вершину* }
  K_finded:=false; { *Ознака закінчення алгоритму* }
  While K_finded do
    begin
      R2S:=[]; { *Очистка множини околу 2-го порядку* }
      R2(MainNode); { *Формуємо окіл 2-го порядку для MainNode* }
      For k=1 to n do { *Цикл по вершинах графа* }
        begin
          If k in R2S then
            begin { *Вершина належить околу другого порядку* }
              { *Склеювання вершини MainNode з вершиною околу k * }
              Glue(MainNode, k);
              { *Модифікація матриці суміжності після склеювання вершин * }
              Reduce(MainNode, k);
            end;
          end;
          MainNode:= Check_K(MainNode);
          If MainNode=0 then K_finded:=true;
        end;
      End;
    end;
  End;

```

#### 4.10. РЕКУРСИВНА ПРОЦЕДУРА ПОСЛІДОВНОГО РОЗФАРБУВАННЯ

1. Фіксуємо порядок обходу вершин.

2. Ідемо по вершинах, використовуючи такий найменший колір, який не викликає конфліктів.

3. Якщо вже використаний колір вибрати неможливо, то тільки тоді вводимо новий колір.

У процедурі використовується рекурсивний виклик процедури фарбування наступної вершини у випадку успішного фарбування попередньої вершини.

```

Const n=10;
      Cmax=10;
Type
  TA = Array (1..n, 1..n) of Byte;

```



```

    TArr = Array (1..n) of Byte;
Var i:Byte;
    color:TArr;
    A:TA;
    C:Byte;

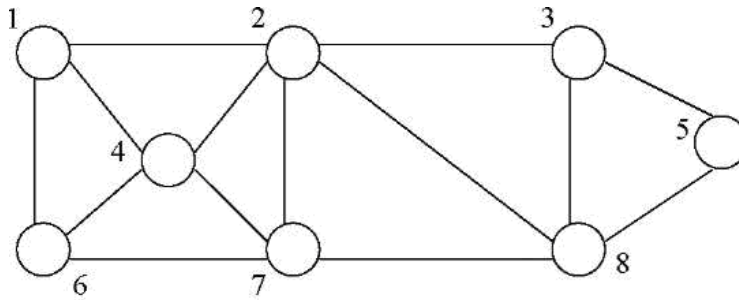
procedure visit(i:Byte);
    Function Nicecolor:Boolean;
    {*Функція пошуку неконфліктної фарби*}
    Var CN:Boolean;
    j:integer;
    Begin
    CN:=true;
    For j=1 to n do
    If (A[j,i]=1) AND (color[j]=c) then CN:=false;
    {*Знайдена суміжна вершина до вершини i. Ця вершина має поточний колір c.
    Тоді колір c є конфліктним*}
    End;
begin
    if i = n + 1 then Print else
    {*Якщо всі вершини розфарбовано, то виводимо результат*}
    begin
    If color[i]=0 then {*Якщо поточна вершина не розфарбована*}
    begin
    for c:=color[i]+1 to Cmax do
    if Nicecolor then
    begin
    color[i]:=c;
    {*Якщо неконфліктний, то розфарбовуємо вершину i фарбою c*}
    visit(i+1);
    {Рекурсивно викликаємо процедуру для розфарбування наступної
    вершини}
    end;
    end;
    end;
    end;

Begin
    i:=1;
    visit(i);
End;

```

#### 4.11. ПРИКЛАД РОБОТИ РЕКУРСИВНОЇ ПРОЦЕДУРИ ПОСЛІДОВНОГО РОЗФАРБУВАННЯ

Розглянемо граф  $G$  та застосуємо рекурсивну процедуру для його розфарбування.



Матриця суміжності  $A$  має такий вигляд

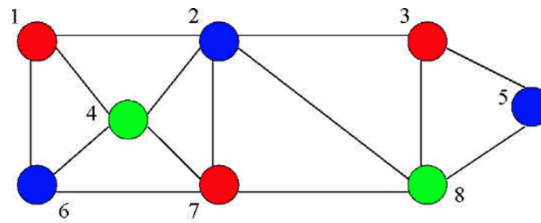
$$A = \begin{pmatrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{matrix} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{pmatrix}$$

Робота алгоритму зведена в таблицю.

Перший стовпець містить виклики процедури Visit, а решта стовпців показує, яка фарба була прийнята, а яка відхилена.

	Червоний	Синій	Зелений
Visit(1)	+		
Visit(2)	-	+	
Visit(3)	+		
Visit(4)	-	-	+
Visit(5)	-	+	
Visit(6)	-	+	
Visit(7)	+		
Visit(8)	-	-	+

Розфарбований граф має вигляд:



#### 4.12. «ЖАДІБНИЙ» АЛГОРИТМ РОЗФАРБУВАННЯ

Нехай дано зв'язний граф  $G(V, E)$ .

1. Задаємо множину  $monochrom := \emptyset$ , куди будемо записувати всі вершини, які можна пофарбувати одним кольором.
2. Переглядаємо всі вершини й виконуємо наступний «жадібний» алгоритм:

##### Procedure Greedy

**For** ( для кожної незафарбованої вершини  $v \in V$  ) **do**

**If**  $v$  не суміжна з вершинами з  $monochrom$  **then**

**begin**

$color(v) := \text{колір};$

$monochrom := monochrom \cup \{v\}$

**End;**

Розглянемо детальніше програмну реалізацію даного алгоритму за умови, що для представлення графа використовують матрицю суміжності.

**Const** N=10; {\*максимальна кількість вершин графа\*}

**Type** V=0..N;

TS=**Set** of V;

TColArr = **Array** (1..N) of V;

TA = **Array** (1..N, 1..N) of Integer;

**Var** ColArr: TColArr; {\*масив номерів фарб для кожної вершини графа\*}

A:TA; {\*матриця суміжності графа\*}

Color:Byte;

AllColored:Boolean;

k:Byte;

**Procedure** Avid(i:Integer);

{\*функція вибору фарби для розфарбування вершини з номером i \*}

**Var** W:TS;

j:Byte;

**function** Check(i):Boolean; {\*Перевірка кольору суміжних вершин\*}

**var** Ch:Boolean;

**begin**

Ch:=true;

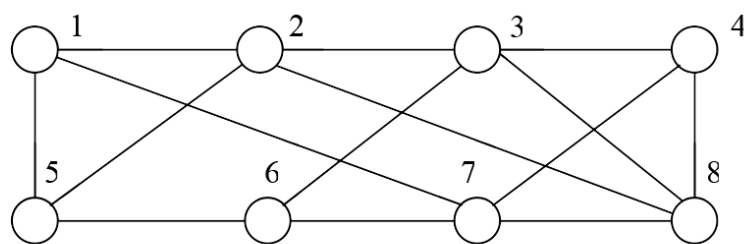
```

For j=1 to n do
  If (A[j,i]=1) then { *Якщо вершина j суміжна з тією, що підлягає
перевірці *}
  If (j in W) then Ch:=false;
  { *Якщо вершина j розфарбована поточним кольором *}
  Check:= Ch;
end;
Begin
  Inc(Color); { *Встановлюємо новий колір *}
  W:=[]; { *Очищаємо множину одноколірних вершин *}
  ColArr[i]:=Color; { *Розфарбовуємо першу вершину новою фарбою *}
  W:=W+[i]; { *Доповнюємо множину одноколірних вершин вершиною i *}
  { *Перевіряємо інші вершини на можливість розфарбування цією фарбою *}
  For k:=1 to n do if ColArr[k]=0 then
    If Check(k) then begin ColArr[k]:=Color; W:=W+[k]; end;
End;
Begin { *Головний цикл *}
  <Вводимо матрицю суміжності графа>
  { *цикл по вершинах графа *}
  Color:=0;
  AllColored:=false; { *Ознака того, що всі вершини розфарбовано *}
  While not AllColored do
    Begin
      AllColored:=true;
      For i=1 to N do If ColArr[i]=0 then
        begin
          { *Знайшли не розфарбовану вершину *}
          AllColored:=false;
          Avid(i); { *процедура жадібного розфарбування *}
        end;
      End;
    <Виводимо результат розфарбування>
End;

```

#### 4.13. ПРИКЛАД РОБОТИ «ЖАДІБНОГО» АЛГОРИТМУ РОЗФАРБУВАННЯ

Розглянемо граф  $G$  та застосуємо до нього «жадібний» алгоритм розфарбування.



Матриця суміжності  $A$  має такий вигляд

$$A = \begin{pmatrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{matrix} & \begin{matrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{matrix} \\ & \begin{matrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{matrix} \\ & \begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{matrix} \\ & \begin{matrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{matrix} \\ & \begin{matrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \end{matrix} \\ & \begin{matrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{matrix} \\ & \begin{matrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{matrix} \\ & \begin{matrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{matrix} \end{pmatrix}$$

Для початку розфарбування вибираємо вершину з номером 1 та розфарбовуємо її в колір 1 (червоний).

Далі відбувається пошук несуміжної вершини з вершиною 1. Якщо така вершина знайдена, то вона також розфарбовується в колір 1 (червоний).

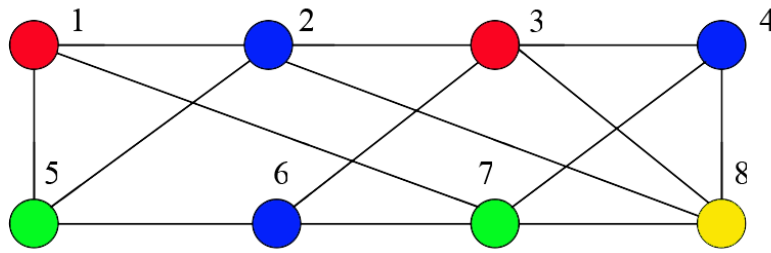
Наступна знайдена для розфарбування кольором 1 вершина повинна бути не суміжною з двома попередніми. Процес продовжується до того часу, поки всі можливості розфарбувати вершини кольором 1 будуть вичерпані.

Після цього вибираємо фарбу кольору 2 (синя) і розфарбовуємо нею вершину з мінімальним номером, яка є не розфарбованою до цього часу. Наступна вершина, яка підходить для розфарбування фарбою 2, повинна бути не суміжною з вершиною, яка була розфарбована кольором 2 (синій) на попередньому кроці. Процес розфарбування фарбою 2 також продовжується до того часу, поки не будуть вичерпані всі можливості розфарбування вершин цієї фарбою.

Перед вибором чергової фарби для розфарбування завжди перевіряємо, чи залишилися ще не розфарбовані вершини. Якщо такі вершини знайдено, то вибираємо чергову фарбу і продовжуємо процес розфарбування.

Якщо ж всі вершини графа розфарбовано, то процес розфарбування жадібним алгоритмом закінчується.

Результат розфарбування «жадібним» алгоритмом графа  $G$  показано на рисунку



#### 4.14. ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ № 4

1. Вивчити основні означення та теореми про розфарбування графів. Створити програму розфарбування графів, що реалізує алгоритм прямого перебору.

2. Набути теоретичні знання по темі «Розфарбування графів». Створити програму розфарбування графів, яка реалізує евристичний алгоритм розфарбування.

3. Вивчити основні означення та теореми про розфарбування графів. Створити програму розфарбування графів, яка реалізує модифікований евристичний алгоритм розфарбування.

4. Набути теоретичні знання по темі «Розфарбування графів». Створити програму розфарбування графів, яка реалізує алгоритм розфарбування методом А. П. Єршова.

5. Вивчити основні означення та теореми про розфарбування графів. Створити програму розфарбування графів за рекурсивною процедурою послідовного розфарбування.

6. Набути теоретичні знання по темі «Розфарбування графів». Створити програму розфарбування графів, яка реалізує «жадібний» алгоритм розфарбування.

#### 4.15. Вимоги до програмного забезпечення:

1. Лабораторна робота виконується з використанням скриптової мови програмування Python.

2. Для написання коду застосувати IDE PyCharm 3 Edu.

3. Для написання GUI застосувати бібліотеку tkinter.

4. Необхідно забезпечити ввід даних з клавіатури та з файлу.

5. Вершини результуючого графа необхідно відобразити розфарбованими у відповідності з результатами роботи алгоритму.

#### Зміст звіту:

1. Титульний лист.
2. Тема завдання.
3. Завдання.

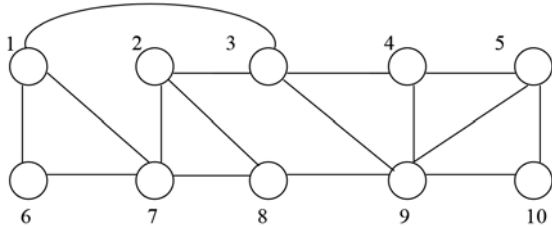
4. Блок-схема програми по п. Б.
5. Роздруківка тексту програми з п. В.
7. Контрольний приклад та результати машинного розрахунку.
8. Висновки по роботі.

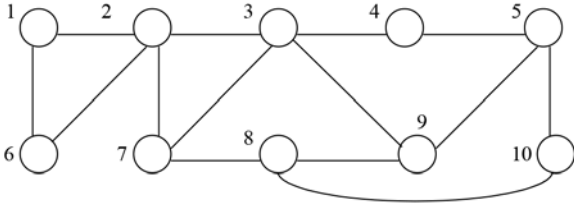
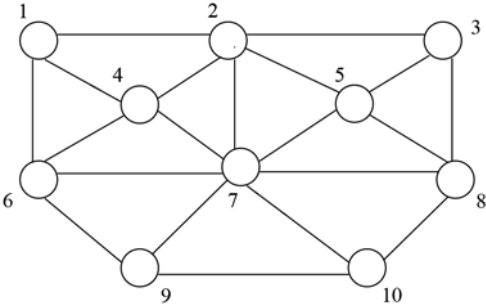
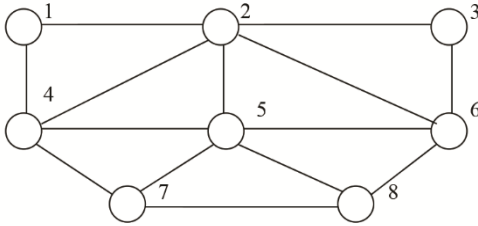
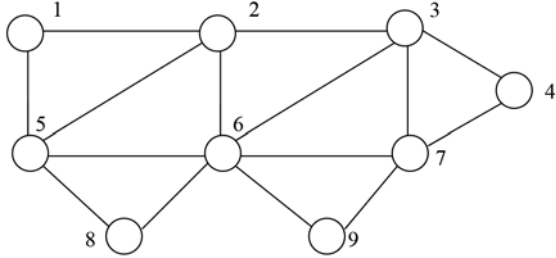
### Контрольні запитання

1. Сформулюйте задачу розфарбовування графа.
2. Який граф називається  $\Gamma$ -хроматичним?
3. Що називається хроматичним числом графа?
4. Як визначаються нижня і верхня оцінки хроматичного числа?
5. Який граф називається планарним? У скільки кольорів його можна розфарбувати?
6. У скільки кольорів можна розфарбувати повний граф?
7. Алгоритм простого перебору.
8. Евристичний алгоритм розфарбовування графа.
9. Модифікований евристичний алгоритм розфарбовування графа.
10. Алгоритм розфарбування за методом А.П. Єршова.
11. Розфарбування за допомогою рекурсивної процедури послідовного розфарбування.
12. «Жадібний» алгоритм розфарбування графів.

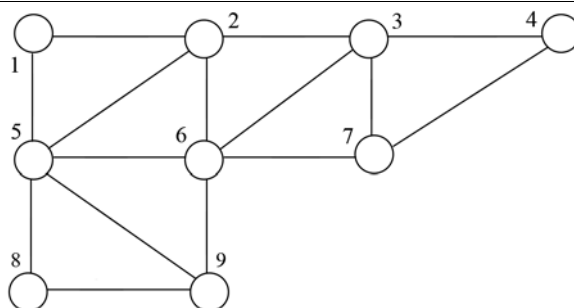
### 4.16. Варіанти для виконання лабораторної роботи

Номер варіанта  $I$  визначається як результат операції  $I = NZK \bmod 6+1$ , де  $NZK$  – номер залікової книжки. При виконанні завдань лабораторної роботи використати такі варіанти задавання графа:

№	Опис варіанта
1	<p>А) Виконати завдання 1 до лабораторної роботи.  Б) Програма повинна дозволяти розфарбування довільного графа.  В) Перевірити роботу програми на даному графі <math>G</math>.</p>  <p>Вивести у графічному режимі розфарбований граф або включити у протокол розфарбований вручну граф за результатами роботи програми.</p>
2	<p>А) Виконати завдання 2 до лабораторної роботи.  Б) Програма повинна дозволяти розфарбування довільного графа.  В) Перевірити роботу програми на даному графі <math>G</math>.</p>

	 <p>Вивести у графічному режимі розфарбований граф або включити у протокол розфарбований вручну граф за результатами роботи програми.</p>
3	<p>А) Виконати завдання 3 до лабораторної роботи.  Б) Програма повинна дозволяти розфарбування довільного графа.  В) Перевірити роботу програми на даному графі <math>G</math>.</p>  <p>Вивести у графічному режимі розфарбований граф або включити у протокол розфарбований вручну граф за результатами роботи програми.</p>
4	<p>А) Виконати завдання 4 до лабораторної роботи.  Б) Програма повинна дозволяти розфарбування довільного графа.  В) Перевірити роботу програми на даному графі <math>G</math>.</p>  <p>Вивести у графічному режимі розфарбований граф або включити у протокол розфарбований вручну граф за результатами роботи програми.</p>
5	<p>А) Виконати завдання 5 до лабораторної роботи.  Б) Програма повинна дозволяти розфарбування довільного графа.  В) Перевірити роботу програми на даному графі <math>G</math>.</p>  <p>Вивести у графічному режимі розфарбований граф або включити у протокол розфарбований вручну граф за результатами роботи програми.</p>
6	<p>А) Виконати завдання 6 до лабораторної роботи.  Б) Програма повинна дозволяти розфарбування довільного графа.  В) Перевірити роботу програми на даному графі <math>G</math>.</p>





Вивести у графічному режимі розфарбований граф, або включити у протокол розфарбований вручну граф за результатами роботи програми.

## Лабораторна робота № 5

**Тема:** «Комбінаторика: перестановки, розміщення, сполучення»

**Мета роботи:** вивчення правил утворення комбінацій множин: *перестановок, розміщень, сполучень*.

**Завдання:** Вивчити алгоритми формування перестановок, сполучень та розбиття. Написати програми для виконання даних алгоритмів.

### Теоретичні основи

#### 5.1. Основні означення

##### Перестановки

*Комбінації з  $n$  елементів, які відрізняються одна від одної тільки порядком елементів, називають перестановками.*

Перестановки позначають символом  $P_n$ , де  $n$  — число елементів, що входять у кожную перестановку.

**Приклад.** Нехай множина  $M$  містить три букви  $A, B, C$ . Складемо всі можливі комбінації із цих букв:  $ABC, ACB, BCA, CAB, CBA, BAC$  (усього 6 комбінацій). Видно, що вони відрізняються одна від одної тільки порядком розташування букв.

*Добуток всіх натуральних чисел від 1 до  $n$  включно називають  $n$ -факторіалом і пишуть:  $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 1) \cdot n$ . Вважають, що  $0! = 1$  і  $n \in N$ . Основна властивість факторіала:  $(n + 1)! = (n + 1) \cdot n!$ .*

Отже, число перестановок обчислюємо за формулою:  $P_n = n!$

##### Розміщення

*Комбінації з  $n$  елементів по  $m$  елементів, які відрізняються одна від одної або самими елементами, або порядком елементів, називають розміщеннями.*

Розміщення позначають символом  $A_n^m$ , де  $n$  — число всіх наявних елементів,  $m$  — число елементів у кожній комбінації. Число розміщень можна обчислити за формулою:

$$A_n^m = n(n-1)(n-2)\dots(n-m+1), \quad \text{де } 0 \leq m \leq n; \quad m, n \in N.$$

Вважають, що  $A_n^0 = 1$ .

**Приклад.** Нехай множина  $M$  містить чотири букви  $A, B, C, D$ . Склавши всі комбінації тільки із двох букв, одержимо:  $AB, AC, AD, BA, BP, BD, CA, CB, CD, DA, DB, DC$ .

Формулу розміщення можна записати у факторіальній формі:

$$A_n^m = \frac{n!}{(n-m)!}.$$

Основні властивості розміщень:

$$1) A_n^{m+1} = A_n^m \cdot (n-m);$$

$$2) A_n^n = P_n = n!.$$

### Сполучення

Сполученнями називають всі можливі комбінації з  $n$  елементів по  $m$ , які відрізняються одна від одної принаймні хоча б одним елементом ( $m, n \in N$  і  $n \geq m$ ).

У загальному випадку число сполучень із  $n$  елементів по  $m$  дорівнює числу розміщень з  $n$  елементів по  $m$ , діленому на число перестановок з  $m$  елементів:

$$C_n^m = \frac{A_n^m}{P_m}.$$
 Використовуючи для кількості розміщень і перестановок факторіальні

формули  $A_n^m = \frac{n!}{(n-m)!}$  і  $P_n = n!$ , одержимо формулу кількості сполучень у

$$\text{вигляді: } C_n^m = \frac{n!}{(n-m)! m!}.$$

$$\text{Основні властивості сполучень: } C_n^{n-m} = \frac{P_n}{P_{n-m} \cdot P_m} = \frac{n!}{(n-m)! m!}; C_n^m = C_n^{n-m}.$$

**Приклад.** Множина  $M$  утворена з чотирьох букв  $A, B, C, D$ . Скласти комбінації з двох букв, що відрізняються хоча б одним елементом.

Маємо  $AB, AC, AD, BP, BD, CD$ . Виходить, що кількість сполучень з чотирьох елементів по два дорівнює 6. Це коротко записують так:  $C_4^2 = 6$ .

### Розміщення з повтореннями

Розміщення з  $n$  елементів по  $k$  відображають упорядковані комбінації різних елементів множини  $M$ ,  $|M|=n$ . Часто доводиться утворювати упорядковані комбінації з повтореннями деяких елементів. Наприклад, з множини  $M = \{A, B\}$

можна утворити вісім комбінацій з трьох елементів:  $AAA, AAB, ABA, BAA, BAB, BBA, ABB, BBB$ . Тут  $n = 2, k = 3$ . Такі упорядковані  $k$ -комбінації називають кортежами довжини  $k$ . Два кортежі (тобто дві загальні комбінації) вважають однаковими, якщо вони мають однакову довжину і на місцях з однаковими номерами стоять однакові елементи.

*Кортеж довжини  $k$  з  $n$  елементів називають розміщенням з повтореннями з  $n$  елементів по  $k$ .*

Кількість кортежів обчислюють за формулою:  $\overline{A_n^k} = n^k$ .

Дійсно, після заповнення першого місця кортежу довжиною  $k$  одним з  $n$  елементів (що можливо зробити  $n$  варіантами) заповнити друге місце кортежу можна знову будь-яким елементом з усієї множини (повторюючи в одному з варіантів елемент, який знаходиться на першому місці), і так далі  $k$  разів. За правилом добутку одержимо, що  $\overline{A_n^k} = \underbrace{n \cdot n \cdot \dots \cdot n}_{k \text{ разів}} = n^k$

## 5.2. Комбінаторні алгоритми

### Алгоритми перестановок

Розглянемо методи генерування послідовностей всіх  $n!$  перестановок множини, що складається з  $n$  елементів. Для цього задану множину представимо у вигляді елементів масиву  $P[1], P[2], \dots, P[n]$

Методи, які будуть нами розглядатися, базуються на застосуванні до масиву  $P[i], i=1, 2, \dots, n$  елементарної операції, що носить назву поелементної транспозиції. Суть операції полягає у обміні даними між елементами масиву  $P[i]$  і  $P[j], 1 \leq i, j \leq n$  за такою схемою:

$$vrem := P[i], P[i] := P[j], P[j] := vrem,$$

де  $vrem$  – деяка допоміжна змінна, що використовується для тимчасового збереження значення елемента масиву  $P[i]$ .

Пояснимо також термін „лексикографічний порядок”. Назва терміну походить від його застосування у словниках. Всі слова у словниках розміщені у лексикографічному порядку.

**Визначення лексикографічного порядку.** Нехай існують перестановки у вигляді послідовностей  $\{x_1, x_2, x_3, \dots, x_n\}, \{y_1, y_2, y_3, \dots, y_n\}, \dots$  однієї і тієї ж множини  $X$ . Говорять, що перестановки з елементів множини  $X$  упорядковані у лексикографічному порядку, якщо

$$\{x_1, x_2, x_3, \dots, x_n\} < \{y_1, y_2, y_3, \dots, y_n\}$$

тоді і тільки тоді, коли для деякого  $k$

$$x_k \leq y_k \text{ і } x_i = y_i \text{ для всіх } i < k.$$

**Визначення антилексикографічного порядку.** Нехай існують перестановки у вигляді послідовностей  $\{x_1, x_2, x_3, \dots, x_n\}, \{y_1, y_2, y_3, \dots, y_n\}, \dots$  однієї і тієї ж множини  $X$ . Говорять, що перестановки з елементів множини  $X$  упорядковані у антилексикографічному порядку, якщо

$$\{x_1, x_2, x_3, \dots, x_n\} < \{y_1, y_2, y_3, \dots, y_n\}$$

тоді і тільки тоді, коли для деякого  $k$

$$x_k > y_k \text{ і } x_i = y_i \text{ для всіх } i < k.$$

**Приклад.** Перестановки множини  $X = \{1, 2, 3\}$  в лексикографічному (а) та антилексикографічному (б) порядку

(а)	(б)
1 2 3	1 2 3
1 3 2	2 1 3
2 1 3	1 3 2
2 3 1	3 1 2
3 1 2	2 3 1
3 2 1	3 2 1

### 5.2.1. Алгоритм побудови перестановок у лексикографічному порядку

Алгоритм формування перестановок у лексикографічному порядку почнемо, наприклад, з початкової перестановки  $(1, 2, \dots, n-1, n)$ . Подальший розвиток алгоритму полягає у переході від перестановки  $(x_1, x_2, \dots, x_{n-1}, x_n)$  до безпосередньо наступної за нею перестановки  $(y_1, y_2, \dots, y_n)$  поки не одержимо найбільшу перестановку  $(n, n-1, \dots, 2, 1)$ . Розглянемо спосіб побудови перестановки  $(y_1, y_2, \dots, y_{n-1}, y_n)$ .

#### (1, 2, 3) Цикл 1

11. Розглядаємо справа наліво перестановку  $x = (x_1, x_2, \dots, x_{n-1}, x_n)$  в пошуках самої правої позиції  $i$  такої, що  $x_i < x_{i+1}$ . Якщо такої позиції не знайдено, то тоді  $x_1 > x_2 > \dots > x_n$ , тобто  $x = (n, n-1, \dots, 1)$ . Дана перестановка є завершальною перестановкою нашого алгоритму.

**Приклад.**  $x = (1, 2, 3)$ .  $x_1 = 1$ ,  $x_2 = 2$ ,  $x_3 = 3$ .

$x_2 < x_3$  оскільки  $2 < 3$ . Отже сама права позиція, для якої виконується  $x_i < x_{i+1}$  визначається індексом  $i = 2$

12. Якщо ж згадану позицію  $i$  знайдено, то  $x_i < x_{i+1} > x_{i+2} > \dots > x_n$ .

**Приклад.**  $x = (1, 2, 3)$  При  $i = 2$  одержуємо  $2 < 3 > \emptyset$  оскільки 3 є останнім елементом кортежу.

13. На наступному кроці алгоритму шукаємо першу позицію  $j$  при переході від позиції  $n$  до позиції  $i$  таку, що  $x_i < x_j$ . Тоді  $i < j$ .

**Приклад.**  $x = (1, 2, 3)$ . Шукана позиція  $j = 3$ , оскільки  $x_2 < x_3$ . Тоді  $2 < 3$ .

14. Далі виконуємо операцію транспозиції над елементами  $x_i$  та  $x_j$ , в результаті якої одержуємо перестановку  $x' = (x'_1, x'_2, \dots, x'_n)$ .

**Приклад.**  $x = (1, 2, 3)$ . Виконали транспозицію елементів  $x_2$  і  $x_3$ . Одержали  $x = (1, 3, 2)$

15. В цій перестановці частину елементів  $x_{i+1}, \dots, x_{n-1}, x_n$  перевертаємо, тобто міняємо порядок їх слідування на протилежний. Одержана таким чином перестановка є перестановкою  $y = (y_1, y_2, \dots, y_n)$ . Саме ця перестановка є наступною в лексикографічному порядку слідування перестановок.

**Приклад.**  $x = (1, 3, 2)$ . В даному випадку необхідно перевернути терм, що складається з одного елемента ( $x_3$ ), що нічого не змінює. Тому одержуємо  $x = (1, 3, 2)$ .

Переходимо до циклу 2 алгоритму.

### **(1, 3, 2) Цикл 2.**

21. Розглядаємо справа наліво перестановку  $x = (x_1, x_2, \dots, x_{n-1}, x_n)$  в пошуках самої правої позиції  $i$  такої, що  $x_i < x_{i+1}$ . Якщо такої позиції не знайдено, то тоді  $x_1 > x_2 > \dots > x_n$ , тобто  $x = (n, n-1, \dots, 1)$ . Дана перестановка є завершальною перестановкою нашого алгоритму.

**Приклад.**  $x = (1, 3, 2)$ .  $x_1 = 1$ ,  $x_2 = 3$ ,  $x_3 = 2$ .

$x_1 < x_2$ , оскільки  $1 < 3$ . Отже, сама права позиція, для якої виконується  $x_i < x_{i+1}$ , визначається індексом  $i = 1$

22. Якщо ж згадану позицію  $i$  знайдено, то  $x_i < x_{i+1} > x_{i+2} > \dots > x_n$ .

**Приклад.**  $x = (1, 2, 4, 3)$ . При  $i = 1$  одержуємо  $1 < 3 > 2$ .

23. На наступному кроці алгоритму шукаємо першу позицію  $j$  при переході від позиції  $n$  до позиції  $i$  таку, що  $x_i < x_j$ . Тоді  $i < j$ .

**Приклад.**  $x = (1, 3, 2)$ . Шукана позиція  $j = 3$ , оскільки  $x_1 < x_3$ . Тоді  $1 < 2$ .

24. Далі виконуємо операцію транспозиції над елементами  $x_i$  та  $x_j$ , в результаті якої одержуємо перестановку  $x' = (x'_1, x'_2, \dots, x'_n)$ .

**Приклад.**  $x = (1, 3, 2)$ . Виконали транспозицію елементів  $x_1$  і  $x_3$ . Одержали  $x = (2, 3, 1)$ .

25. В цій перестановці частину елементів  $x_{i+1}, \dots, x_{n-1}, x_n$  перевертаємо, тобто міняємо порядок їх слідування на протилежний. Одержана таким чином перестановка є перестановкою  $y = (y_1, y_2, \dots, y_n)$ . Саме ця перестановка є наступною в лексикографічному порядку слідування перестановок.

**Приклад.**  $x = (2, 3, 1)$ . В даному випадку необхідно перевернути терм, що складається з двох елементів  $(x_2, x_3)$ . Одержуємо  $x = (2, 1, 3)$ .

Переходимо до циклу 3 алгоритму.

### **(2, 1, 3) Цикл 3.**

31. Розглядаємо справа наліво перестановку  $x = (x_1, x_2, \dots, x_{n-1}, x_n)$  в пошуках самої правої позиції  $i$  такої, що  $x_i < x_{i+1}$ . Якщо такої позиції не знайдено, то тоді  $x_1 > x_2 > \dots > x_n$ , тобто  $x = (n, n-1, \dots, 1)$ . Дана перестановка є завершальною перестановкою нашого алгоритму.

**Приклад.**  $x = (2, 1, 3)$ .  $x_1 = 2$ ,  $x_2 = 1$ ,  $x_3 = 3$ .

А) Спочатку перевіряємо  $x_2 < x_3 \Rightarrow 1 < 3$ .  $i = 2$ . Справджується, оскільки  $1 < 3$

32. Якщо ж згадану позицію  $i$  знайдено, то  $x_i < x_{i+1} > x_{i+2} > \dots > x_n$ .

**Приклад.**  $x = (2, 1, 3)$ . При  $i = 2$  одержуємо  $1 < 3 > \emptyset$ .

33. На наступному кроці алгоритму шукаємо першу позицію  $j$  при переході від позиції  $n$  до позиції  $i$  таку, що  $x_i < x_j$ . Тоді  $i < j$ .

**Приклад.**  $x = (2, 1, 3)$ . Шукана позиція  $j = 3$ , оскільки  $x_2 < x_3$ . Тоді  $1 < 3$ .

34. Далі виконуємо операцію транспозиції над елементами  $x_i$  та  $x_j$ , в результаті якої одержуємо перестановку  $x' = (x'_1, x'_2, \dots, x'_n)$ .

**Приклад.**  $x = (2, 1, 3)$ . Виконали транспозицію елементів  $x_2$  і  $x_3$ . Одержали  $x = (2, 3, 1)$ .

35. В цій перестановці частину елементів  $x_{i+1}, \dots, x_{n-1}, x_n$  перевертаємо, тобто міняємо порядок їх слідування на протилежний. Одержана таким чином перестановка є перестановкою  $y = (y_1, y_2, \dots, y_n)$ . Саме ця перестановка є наступною в лексикографічному порядку слідування перестановок.

**Приклад.**  $x = (2, 3, 1)$ . В даному випадку необхідно перевернути терм, що складається з одного елемента  $(x_3)$ . Одержуємо  $x = (2, 3, 1)$ .

Переходимо до циклу 4 алгоритму.

#### **(2, 3, 1) Цикл 4.**

41. Розглядаємо справа наліво перестановку  $x = (x_1, x_2, \dots, x_{n-1}, x_n)$  в пошуках самої правої позиції  $i$  такої, що  $x_i < x_{i+1}$ . Якщо такої позиції не знайдено, то тоді  $x_1 > x_2 > \dots > x_n$ , тобто  $x = (n, n-1, \dots, 1)$ . Дана перестановка є завершальною перестановкою нашого алгоритму.

**Приклад.**  $x = (2, 3, 1)$ .  $x_1 = 2$ ,  $x_2 = 3$ ,  $x_3 = 1$ .

А) Спочатку перевіряємо  $x_2 < x_3 \Rightarrow 3 > 1$ . Не справджується, оскільки  $3 > 1$ .

Б) Далі перевіряємо  $x_1 < x_2 \Rightarrow 2 < 3$ . Справджується, оскільки  $2 < 3$ .

Отже,  $i = 1$ .

42. Якщо ж згадану позицію  $i$  знайдено, то  $x_i < x_{i+1} > x_{i+2} > \dots > x_n$ .

**Приклад.**  $x = (2, 3, 1)$ . При  $i = 1$  одержуємо  $2 < 3 > 1$ .

43. На наступному кроці алгоритму шукаємо першу позицію  $j$  при переході від позиції  $n$  до позиції  $i$  таку, що  $x_i < x_j$ . Тоді  $i < j$ .

**Приклад.**  $x = (2, 3, 1)$ . Шукана позиція  $j = 2$ , оскільки  $x_1 < x_2$ . Тоді  $2 < 3$ .

44. Далі виконуємо операцію транспозиції над елементами  $x_i$  та  $x_j$ , в результаті якої одержуємо перестановку  $x' = (x'_1, x'_2, \dots, x'_n)$ .

**Приклад.**  $x = (2, 3, 1)$ . Виконали транспозицію елементів  $x_2$  і  $x_3$ . Одержали  $x = (3, 2, 1)$ .

45. В цій перестановці частину елементів  $x_{i+1}, \dots, x_{n-1}, x_n$  перевертаємо, тобто міняємо порядок їх слідування на протилежний. Одержана таким чином перестановка є перестановкою  $y = (y_1, y_2, \dots, y_n)$ . Саме ця перестановка є наступною в лексикографічному порядку слідування перестановок.

**Приклад.**  $x = (3, 2, 1)$ . В даному випадку необхідно перевернути терм, що складається з двох елементів  $(x_2, x_3)$ . Одержуємо  $x = (3, 1, 2)$ .

Переходимо до циклу 5 алгоритму.

#### **(3, 1, 2) Цикл 5.**

51. Розглядаємо справа наліво перестановку  $x = (x_1, x_2, \dots, x_{n-1}, x_n)$  в пошуках самої правої позиції  $i$  такої, що  $x_i < x_{i+1}$ . Якщо такої позиції не знайдено, то тоді  $x_1 > x_2 > \dots > x_n$ , тобто  $x = (n, n-1, \dots, 1)$ . Дана перестановка є завершальною перестановкою нашого алгоритму.

**Приклад.**  $x = (3, 1, 2)$ .  $x_1 = 3$ ,  $x_2 = 1$ ,  $x_3 = 2$ .



А) Спочатку перевіряємо  $x_2 < x_3 \Rightarrow 1 < 2$ . Справджується, оскільки  $1 < 2$ .

Отже,  $i = 2$ .

52. Якщо ж згадану позицію  $i$  знайдено, то  $x_i < x_{i+1} > x_{i+2} > \dots > x_n$ .

**Приклад.**  $x = (3, 1, 2)$ . При  $i = 2$  одержуємо  $1 < 2 > \emptyset$ .

53. На наступному кроці алгоритму шукаємо першу позицію  $j$  при переході від позиції  $n$  до позиції  $i$  таку, що  $x_i < x_j$ . Тоді  $i < j$ .

**Приклад.**  $x = (3, 1, 2)$ . Шукана позиція  $j = 3$ , оскільки  $x_2 < x_3$ . Тоді  $1 < 2$ .

54. Далі виконуємо операцію транспозиції над елементами  $x_i$  та  $x_j$ , в результаті якої одержуємо перестановку  $x' = (x'_1, x'_2, \dots, x'_n)$ .

**Приклад.**  $x = (3, 1, 2)$ . Виконали транспозицію елементів  $x_2$  і  $x_3$ . Одержали  $x = (3, 2, 1)$

55. В цій перестановці частину елементів  $x_{i+1}, \dots, x_{n-1}, x_n$  перевертаємо, тобто міняємо порядок їх слідування на протилежний. Одержана таким чином перестановка є перестановкою  $y = (y_1, y_2, \dots, y_n)$ . Саме ця перестановка є наступною в лексикографічному порядку слідування перестановок.

**Приклад.**  $x = (3, 2, 1)$ . В даному випадку необхідно перевернути терм, що складається з одного елемента  $(x_3)$ . Одержуємо  $x = (3, 2, 1)$ .

Переходимо до циклу 6 алгоритму.

**(3, 2, 1) Цикл 6.**

61. Розглядаємо справа наліво перестановку  $x = (x_1, x_2, \dots, x_{n-1}, x_n)$  в пошуках самої правої позиції  $i$  такої, що  $x_i < x_{i+1}$ . Якщо такої позиції не знайдено, то тоді  $x_1 > x_2 > \dots > x_n$ , тобто  $x = (n, n-1, \dots, 1)$ . Дана перестановка є завершальною перестановкою нашого алгоритму.

**Приклад.**  $x = (3, 2, 1)$ .  $x_1 = 3$ ,  $x_2 = 2$ ,  $x_3 = 1$ .

Не знайдено випадку  $x_i < x_{i+1}$ . Алгоритм завершено.

### 5.2.2. Блок-схема алгоритму побудови перестановок у лексикографічному порядку

#### Блок 1.

Інтерфейс вводу початкових даних:

$n$  – кількість елементів масиву для виконання перестановок у лексикографічному порядку.

$s$  – кількість перестановок, які необхідно виконати.

$P$  – початкові значення елементів масиву для виконання перестановок.

#### Блоки 2-3.

Перевірка отримання кількості замовлених перестановок  $s$ .

#### Блоки 4-7.

Відбувається попереднє встановлення змінних, пошук самої правої позиції  $i$  такої, що  $x_i < x_{i+1}$  та закінчення алгоритму у випадку, коли  $i$  не знайдено.

#### **Блоки 8-10.**

Якщо позиція  $i$  знайдена, то відбувається пошук позиції  $j$ , починаючи від позиції  $n$  до позиції  $i$  такої, що  $x_i < x_j$ .

#### **Блок 11.**

Якщо знайдено  $i$  та  $j$ , то виконуємо операцію транспозиції над елементами  $x_i$  та  $x_j$ .

#### **Блоки 12-17.**

Частину елементів  $x_{i+1}, \dots, x_{n-1}, x_n$  перевертаємо, тобто міняємо порядок їх слідування на протилежний. Роздруківка перестановки.

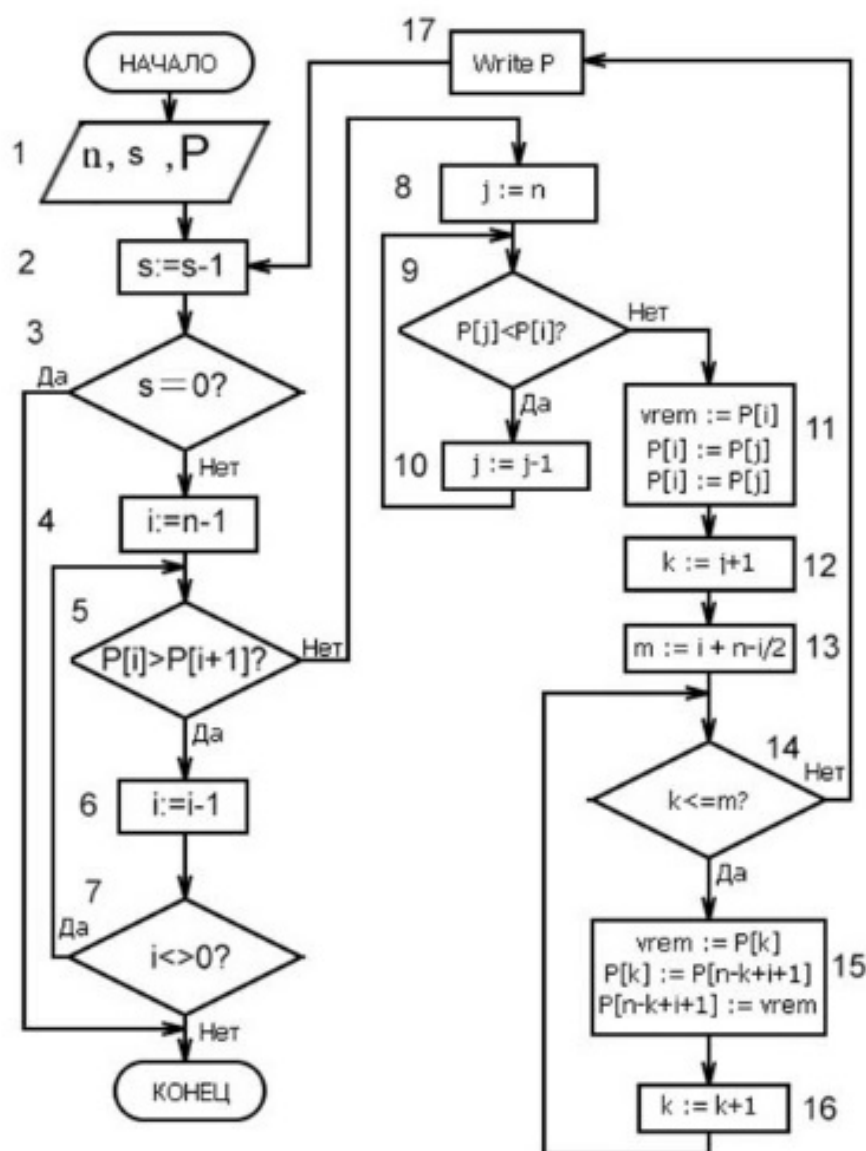


Рис. 5.1. Блок-схема алгоритму перестановок в лексикографічному порядку  
5.2.3. Алгоритм генерації двійкових векторів довжини  $n$ .

Алгоритм породжує всі двійкові вектори  $b = (b_{n-1}, b_{n-1}, \dots, b_1, b_0)$  довжини  $n$  в лексикографічному порядку, починаючи з найменшого елемента.

Будемо використовувати масив  $b[n], b[n-1], \dots, b[1], b[0]$ , установивши  $b[n] := 0$ .

**Приклад.**  $b = (0, 0, 0)$ .  $n = 2$   $b[2] = 0$ ,  $b[1] = 0$ ,  $b[0] = 0$

#### **(0,0,0) Цикл 1.**

1.1. Переглядаючи справа наліво, знаходимо першу позицію  $b[i]$  таку, що  $b[i] = 0$ .

**Приклад.**  $b = (0, 0, 0)$ ,  $i = 0$ ,  $b[0] := 0$

1.2. Записуємо  $b[i] := 1$ , а всі елементи  $b[j]$ ,  $j < i$ , що розміщені праворуч від  $b[i]$ , задаємо рівними 0.

**Приклад.**  $b[0] := 1$ . Оскільки  $i = 0$ , то розряди справа відсутні.

#### **(0,0,1) Цикл 2.**

2.1. Переглядаючи справа наліво, знаходимо першу позицію  $b[i]$  таку, що  $b[i] = 0$ .

**Приклад.**  $b = (0, 0, 1)$ ,  $i = 1$ ,  $b[1] = 0$ .

2.2. Записуємо  $b[i] := 1$ , а всі елементи  $b[j]$ ,  $j < i$ , що розміщені праворуч від  $b[i]$ , задаємо рівними 0.

**Приклад.**  $b[1] := 1$ ,  $b[0] := 0$ .

#### **(0,1,0) Цикл 3.**

1.1. Переглядаючи справа наліво, знаходимо першу позицію  $b[i]$  таку, що  $b[i] = 0$ .

**Приклад.**  $b = (0, 1, 0)$ ,  $i = 0$ ,  $b[0] := 0$

1.2. Записуємо  $b[i] := 1$ , а всі елементи  $b[j]$ ,  $j < i$ , що розміщені праворуч від  $b[i]$ , задаємо рівними 0.

**Приклад.**  $b[0] := 1$ . Оскільки  $i = 0$ , то розряди справа відсутні.

#### **(0,1,1) Цикл 4.**

2.1. Переглядаючи справа наліво, знаходимо першу позицію  $b[i]$  таку, що  $b[i] = 0$ .

**Приклад.**  $b = (0, 1, 1)$ ,  $i = 1$ ,  $b[2] = 0$ .

2.2. Записуємо  $b[i] := 1$ , а всі елементи  $b[j]$ ,  $j < i$ , що розміщені праворуч від  $b[i]$ , задаємо рівними 0.

Для всіх породжуваних послідовностей елемент  $b[n]$  не змінюється, за винятком генерації останнього вектора  $(1, 1, \dots, 1)$ ,  $i = n$ . Рівність  $b[n] = 1$  є умовою зупинки алгоритму.

**Приклад.**  $b[2] := 1$ . Оскільки  $n = 2$ , то елемент масиву  $b[n] = 1$  є ознакою закінчення алгоритму.

### **5.2.4. Блок-схема алгоритму генерації двійкових векторів довжини $n$ .**

**Блок 1.** Ввод початкових параметрів алгоритму.

$n$  – кількість елементів двійкового вектора.

$m$  – кількість двійкових векторів послідовності, які потрібно згенерувати за допомогою даного алгоритму.

$(b[n-1], b[n-2], \dots, b[0])$  – початковий вектор послідовності векторів.

**Блок 2.**  $b[n] := 0$ . Встановлення початкового стану для ознаки закінчення роботи алгоритму внаслідок досягнення максимального вектора послідовності.

**Блок 3.** Перевірка ознаки досягнення максимального вектора послідовності.

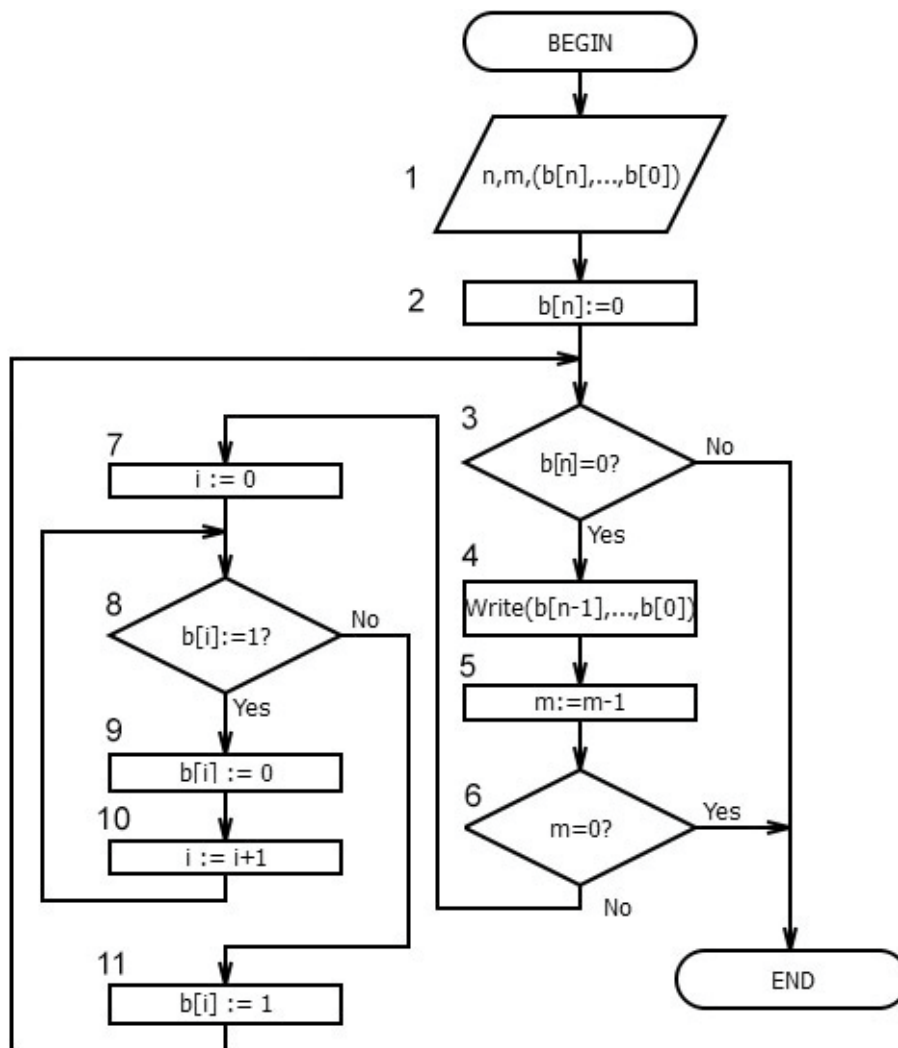


Рис. 5.2. Блок-схема алгоритму генерації двійкових векторів довжини  $n$ .

**Блок 4.** Роздруківка чергового вектора послідовності.

**Блок 5.** Зміна значення лічильника кількості векторів, заданих для генерації.

**Блок 6.** Перевірка ознаки кількості згенерованих векторів.

**Блок 7.** Встановити в початкове значення індекс для пошуку.

**Блоки 8, 9, 10.** Пошук справа наліво першої нульової позиції з одночасним обнулінням переглянутих одиничних елементів.

**Блок 11.** Встановлення в 1 знайденої в результаті пошуку справа наліво нульової позиції.

Після виконання блоку 11 завжди відбуваються перевірки на досягнення максимального значення двійкового вектору та на досягнення заданої кількості згенерованих векторів.

### 5.2.5. Алгоритм генерації підмножин заданої множини

Нехай дано множину  $A = \{a_0, a_1, \dots, a_i, \dots, a_{n-1}\}$ . Введемо фіктивний елемент  $a_n \in A$ . Будемо далі розглядати алгоритм пошуку підмножин  $B^i$  множини  $A = \{a_0, a_1, \dots, a_i, \dots, a_{n-1}, a_n\}$ . Кожна наступна підмножина  $B^i$  формується на основі попередньої підмножини  $B^{i-1}$ .

**Приклад.**  $A = \{a_0, a_1, a_2\}$   $n = 2$

Першою завжди вибираємо пусту підмножину  $B^0 = \emptyset$ .

$\emptyset$  **Цикл 1.**

1.1. Переглядаємо елементи множини в порядку збільшення індексу  $i$  з метою пошуку відсутнього в поточній підмножині елемента.

**Приклад.**  $a_0 \notin B^0$ .

1.2. Для формування наступної підмножини включаємо в попередню підмножину знайдений відсутній елемент та виключаємо з попередньої підмножини всі елементи з індексом, що не перевищує індекс включеного елемента.

**Приклад.**  $B^1 := B^0 \cup a_0$ . Оскільки  $B^0$  – це пуста множина, то виключати нічого.

$B^1 = \{a_0\}$  **Цикл 2.**

2.1. Переглядаємо елементи множини в порядку збільшення індексу  $i$  з метою пошуку відсутнього в поточній підмножині елемента.

**Приклад.**  $a_1 \notin B^1$ .

2.2. Для формування наступної підмножини включаємо в попередню підмножину знайдений відсутній елемент та виключаємо з попередньої підмножини всі елементи з індексом, що не перевищує індекс включеного елемента.

**Приклад.** Включаємо  $a_1$ :  $B^2 := B^1 \cup a_1$ . Виключаємо  $a_0$ :  $B^2 := B^2 \setminus \{a_0\}$ .

$B^2 = \{a_1\}$  **Цикл 3.**

2.1. Переглядаємо елементи множини в порядку збільшення індексу  $i$  з метою пошуку відсутнього в поточній підмножині елемента.

**Приклад.**  $a_0 \notin B^2$ .

2.2. Для формування наступної підмножини включаємо в попередню підмножину знайдений відсутній елемент та виключаємо з попередньої підмножини всі елементи з індексом, що не перевищує індекс включеного елемента.

**Приклад.** Включаємо  $a_0$ :  $B^3 := B^2 \cup a_0$ . Оскільки індекс  $i = 0$  є мінімальним, то в даному випадку немає виключень з підмножини  $B^3$ .  $B^3 = \{a_0, a_1\}$

#### Цикл 4.

2.1. Переглядаємо елементи множини в порядку збільшення індексу  $i$  з метою пошуку відсутнього в поточній підмножині елемента.

**Приклад.**  $a_2 \notin B^3$ .

2.2. Для формування наступної підмножини включаємо в попередню підмножину знайдений відсутній елемент та виключаємо з попередньої підмножини всі елементи з індексом, що не перевищує індекс включеного елемента.

**Приклад.** Оскільки елемент  $a_2$  є фіктивним елементом, введеним в множину як ознаку закінчення роботи алгоритму, то вибір цього елемента означає, що всі підмножини  $B^i$  множини  $A$  згенеровано. Тому в даному випадку алгоритм закінчує свою роботу.

### 5.2.6. Блок-схема алгоритму генерації підмножин заданої множини

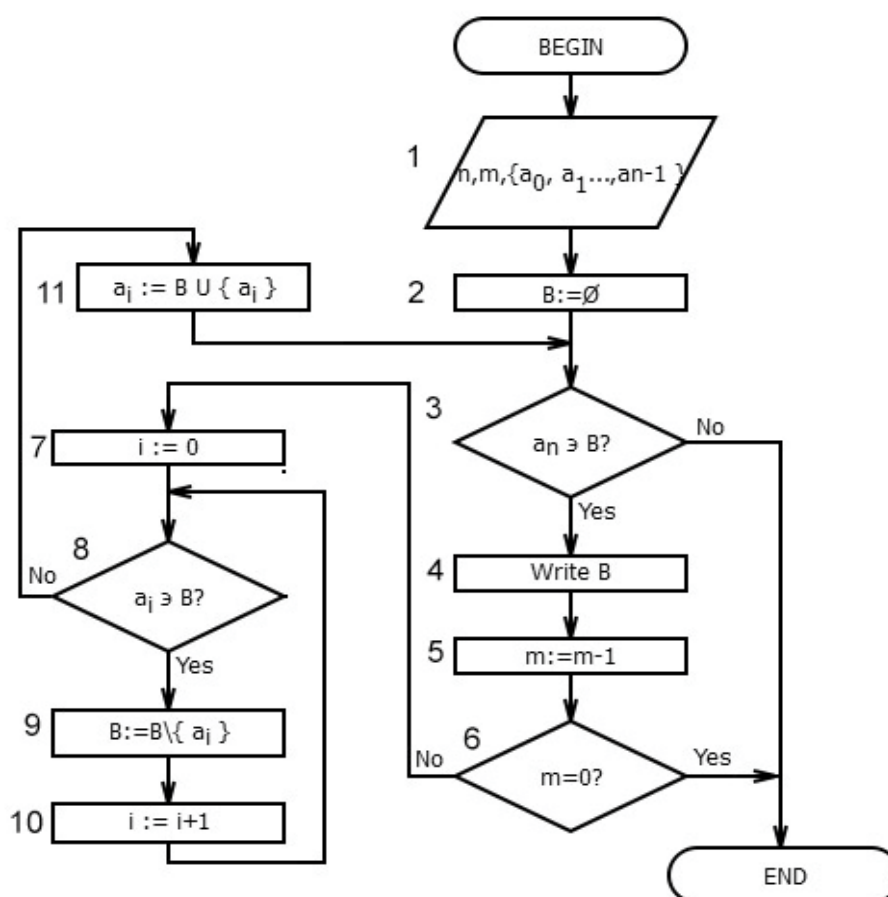


Рис. 5.3. Блок-схема алгоритму генерації підмножин заданої множини

**Блок 1.** Ввід початкових параметрів алгоритму.  $n$  – кількість елементів множини.  $m$  – кількість підмножин, які потрібно згенерувати.  $A$  – початкова множина.

**Блок 2.**  $B := \emptyset$ . Встановлення початкового стану для ознаки закінчення роботи алгоритму після генерації послідовності всіх підмножин.

**Блок 3.** Перевірка ознаки досягнення максимальної кількості підмножин.

**Блок 4.** Роздруківка чергової підмножини.

**Блок 5.** Зміна значення лічильника кількості підмножин, заданих для генерації.

**Блок 6.** Перевірка ознаки кількості згенерованих підмножин.

**Блок 7.** Встановити в початкове значення індекс для пошуку.

**Блоки 8, 9, 10.** Пошук першого елемента у напрямку зростання індексу, який не входить в поточну підмножину з видаленням знайдених елементів.

**Блок 11.** Об'єднання знайденого елемента з поточною підмножиною.

### 5.2.7. Перший алгоритм генерації коду Грея

Нехай  $b_1 b_2 \dots b_n$  – деяке двійкове число.

Код Грея числа одержують шляхом зсуву цього числа на один розряд вправо, відкидання самого правого ( $n$ -го розряду), порозрядного додавання по модулю два із цим же числом без зсуву:

$b_1 b_2 b_3 \dots b_{n-1} b_n$  Таким чином, кожний результуючий  
 $\oplus b_1 b_2 b_3 \dots b_{n-1} \cancel{b_n}$  розряд  $c_i$  одержують за формулою  
 $c_1 c_2 c_3 \dots c_{n-1} c_n$   $c_i = b_i \oplus b_{i-1}$ , вважаючи, що  $b_0 = 0$ .

**Приклад.** Розглянемо порядок генерації коду Грея для трьохрозрядних двійкових чисел:

$i$	Двійкове число	Операція	Код Грея
0	000	$000 \oplus 00 = 000$	000
1	001	$001 \oplus 00 = 001$	001
2	010	$010 \oplus 01 = 011$	011
3	011	$011 \oplus 01 = 010$	010
4	100	$100 \oplus 10 = 110$	110
5	101	$101 \oplus 10 = 111$	111
6	110	$110 \oplus 11 = 101$	101
7	111	$111 \oplus 11 = 100$	100

Виберемо початковий код  $(b_1 b_2 b_3) = 000$ .

Алгоритм містить цикл, кожний прохід якого забезпечує перетворення двійкового коду в код Грея.

#### (000) Цикл 1.

1.1. Виконуємо зсув вправо на один двійковий розряд, відкидаючи зсунутий правий разряд та приписуючи 0 зліва.

$(g_0 g_1 g_2) := 000 \text{ shr } 1$ . Звідси одержуємо  $(g_0 g_1 g_2) = 000$ .

1.2. Виконуємо порозрядне додавання по модулю 2 двох чисел:  $(b_1 b_2 b_3)$  та  $(g_0 g_1 g_2)$ :  $(g_1 g_2 g_3) := (b_1 b_2 b_3) \text{ xor } (g_0 g_1 g_2)$  Звідси  $(g_1 g_2 g_3) := 000$

#### (001) Цикл 2.

2.1. Виконуємо зсув вправо на один двійковий розряд, відкидаючи зсунутий правий розряд та приписуючи 0 зліва.

$(g_0 g_1 g_2) := 001 \text{ shr } 1$ . Звідси одержуємо  $(g_0 g_1 g_2) = 000$ .

2.2. Виконуємо порозрядне додавання по модулю 2 двох чисел:  $(b_1 b_2 b_n)$  та  $(g_0 g_1 g_2)$ .  $(g_1 g_2 g_3) := (b_1 b_2 b_3) \text{ xor } (g_0 g_1 g_2)$  Звідси  $(g_1 g_2 g_3) := 001$

### **(010) Цикл 3.**

3.1. Виконуємо зсув вправо на один двійковий розряд, відкидаючи зсунутий правий розряд та приписуючи 0 зліва.

$(g_0 g_1 g_2) := 010 \text{ shr } 1$ . Звідси одержуємо  $(g_0 g_1 g_2) = 001$ .

3.2. Виконуємо порозрядне додавання по модулю 2 двох чисел:  $(b_1 b_2 b_n)$  та  $(g_0 g_1 g_2)$ .  $(g_1 g_2 g_3) := (b_1 b_2 b_3) \text{ xor } (g_0 g_1 g_2)$   $011 := 010 \text{ xor } 001$

Звідси  $(g_1 g_2 g_3) := 011$

### **(011) Цикл 4.**

3.1. Виконуємо зсув вправо на один двійковий розряд, відкидаючи зсунутий правий розряд та приписуючи 0 зліва.

$(g_0 g_1 g_2) := 011 \text{ shr } 1$ . Звідси одержуємо  $(g_0 g_1 g_2) = 001$ .

3.2. Виконуємо порозрядне додавання по модулю 2 двох чисел:  $(b_1 b_2 b_n)$  та  $(g_0 g_1 g_2)$ .  $(g_1 g_2 g_3) := (b_1 b_2 b_3) \text{ xor } (g_0 g_1 g_2)$   $010 := 011 \text{ xor } 001$

Звідси  $(g_1 g_2 g_3) := 010$

### **(100) Цикл 5.**

3.1. Виконуємо зсув вправо на один двійковий розряд, відкидаючи зсунутий правий розряд та приписуючи 0 зліва.

$(g_0 g_1 g_2) := 100 \text{ shr } 1$ . Звідси одержуємо  $(g_0 g_1 g_2) = 010$ .

3.2. Виконуємо порозрядне додавання по модулю 2 двох чисел:  $(b_1 b_2 b_n)$  та  $(g_0 g_1 g_2)$ .  $(g_1 g_2 g_3) := (b_1 b_2 b_3) \text{ xor } (g_0 g_1 g_2)$   $110 := 100 \text{ xor } 010$

Звідси  $(g_1 g_2 g_3) := 110$

Максимально для трьохрозрядного числа виконуємо  $2^3 = 8$  циклів.

## **5.2.8. Блок-схема першого алгоритму генерації коду Грея**

**Блок 1.** Ввід початкових параметрів алгоритму.

$n$  – кількість розрядів числа у двійковій системі числення.

$m$  – кількість чисел у коді Грея, які потрібно згенерувати.

$(b_1 b_2 \dots b_n)$  початкове число у двійковій системі числення.

**Блок 2.** Головний цикл генерації чисел у коді Грея, починаючи з числа

$(b_1 b_2 \dots b_n)$  у двійковій системі числення та закінчуючи числом  $2^n - 1$ , яке є максимальним  $n$  – розрядним числом.



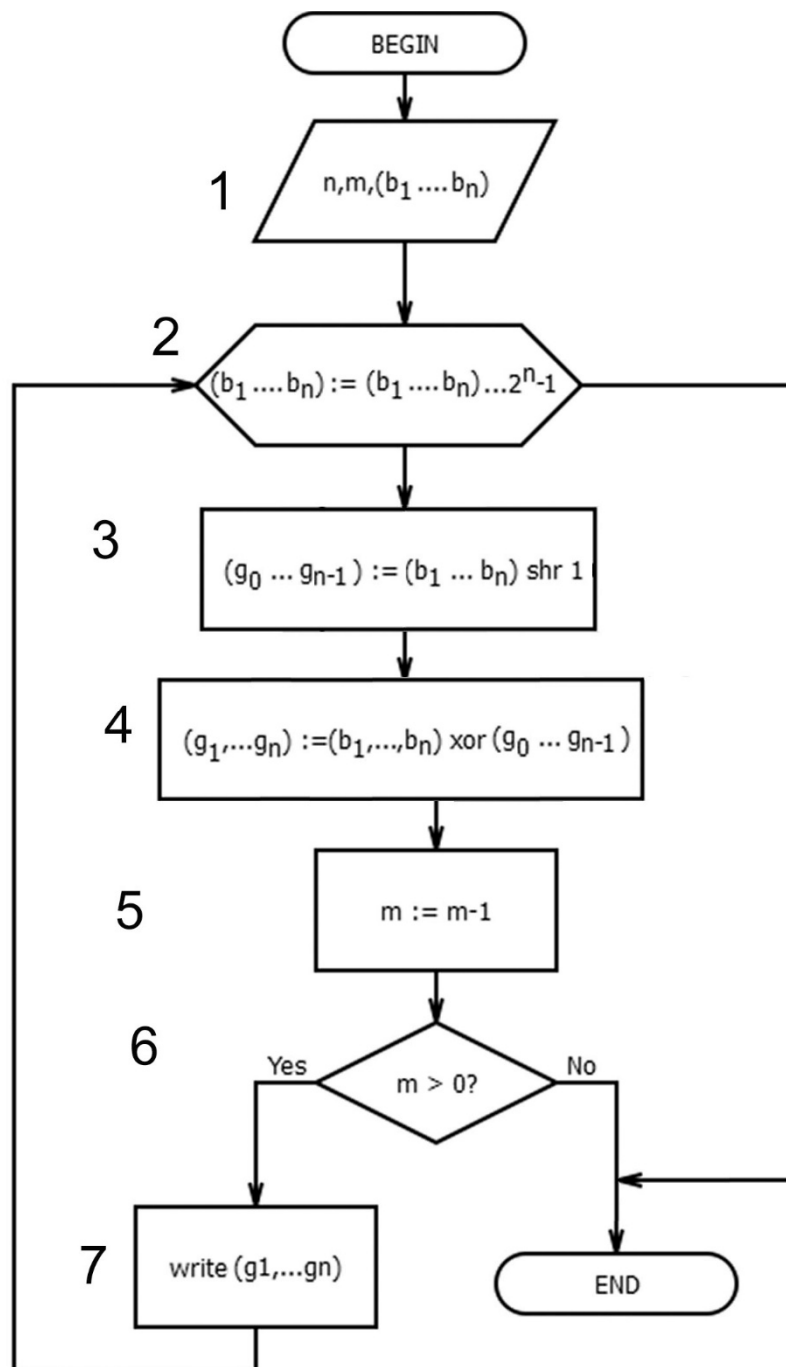


Рис. 5.4. Блок-схема першого алгоритму генерації коду Грея

**Блок 3.** Операція зсуву вправо на один розряд числа  $(b_1 \dots b_n)$ . Результатом зсуву є число  $(g_0 g_1 \dots g_{n-1})$ .

**Блок 4.** Операція порозрядного додавання по модулю два числа  $(b_1 \dots b_n)$  та числа  $(g_0 g_1 \dots g_{n-1})$ . Результатом цієї операції є число  $(g_1 g_2 \dots g_n)$ , яке є числом у коді Грея, що відповідає числу  $(b_1 b_2 \dots b_n)$  у двійковій системі числення.

**Блок 5.** Зміна значення лічильника кількості чисел, заданих для генерації.

**Блок 6.** Перевірка ознаки кількості згенерованих чисел.

**Блок 7.** Роздруківка чергового числа у коді Грея.

### 5.2.9. Другий алгоритм генерації коду Грея

Другий алгоритм генерації коду Грея містить наступні кроки:

1. У якості базової використовуємо двохранрядну послідовність: 00,01,11,10.
  2. Будуємо трьоххранрядну послідовність:
    - 2а. Припишемо до 00,01,11,10 праворуч 0: 000,010,110,100.
    - 2б. Переставимо елементи 00,01,11,10 у зворотному порядку: 10,11,01,00.
    - 2в. До елементів 10,11,01,00 припишемо праворуч 1: 101,111,011,001.
    - 2г. Об'єднаємо послідовності з п.2а й п.2в:  
000, 010,110,100,101,111,011,001.
  3. Для одержання чотирьоххранрядної послідовності перейдемо до п.1 алгоритму, замінивши двохранрядну послідовність послідовністю, отриманою в п. 2г. даного алгоритму.
  4. Повторюючи дії  $n - 2$  рази, одержимо  $n$  – хранрядний код Грея.
- Реалізація даного алгоритму містить два вкладені цикли.  
У зовнішньому циклі відбувається нарощування хранрядів згенерованого коду Грея, а у внутрішньому циклі формується послідовність всіх чисел у кодi Грея в рамках даної кількості хранрядів.

### 5.2.10. Блок-схема другого алгоритму генерації коду Грея

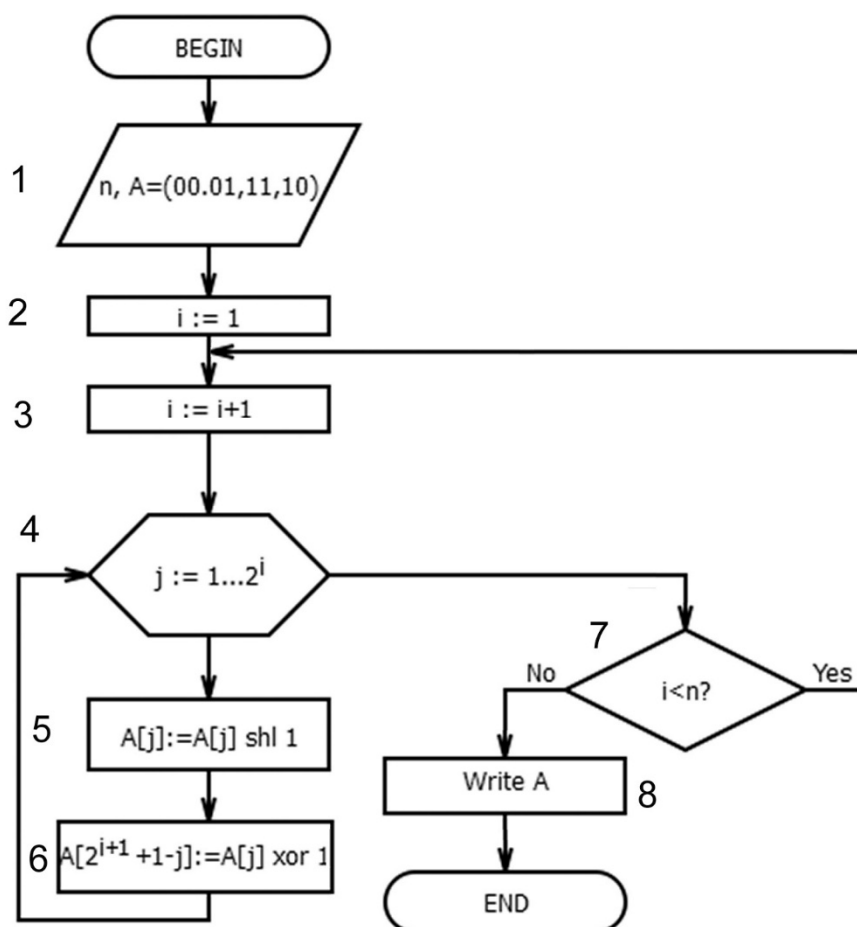


Рис. 5.5. Блок-схема другого алгоритму генерації коду Грея

**Блок 1.** Ввід початкових параметрів алгоритму.

$n$  – кількість розрядів чисел у коді Грея, які потрібно згенерувати.

$A = (00, 01, 11, 10)$  - початкова послідовність чисел, яка відповідає двохранрядним числам у коді Грея.

**Блок 2.** Початкова установка лічильника поточної кількості розрядів чисел, що генеруються алгоритмом у коді Грея.

**Блок 3.** Лічильник кількості розрядів. Алгоритм починає свою роботу з початкової кількості розрядів, яка дорівнює 2.

**Блок 4.** Цикл генерації  $(i+1)$ -розрядної послідовності чисел у коді Грея на основі  $i$  – розрядної послідовності.

**Блок 5.** Зсув вліво на один розряд чисел початкової послідовності у коді Грея.

Операція аналогічна множенню даного числа на 2.

Наприклад:  $0100 * 0010 = 1000$  та  $0100 \text{shl} 1 = 1000$ . Ця операція дозволяє сформувати першу половину  $(i+1)$  – розрядної послідовності чисел у коді Грея.

**Блок 6.** До числа, сформованого у попередньому блоці, додаємо 1 в молодший розряд. Одержані числа записуємо, починаючи з кінця  $(i+1)$  – розрядної послідовності. Таким чином формуємо другу половину  $(i+1)$  – розрядної послідовності чисел у коді Грея.

**Блок 7.** Перевірка ознаки досягнення розрядності  $n$  послідовності чисел у коді Грея.

**Блок 8.** Роздруківка послідовності  $n$ -розрядних чисел у коді Грея.

#### 5.2.11. Перший алгоритм генерації підмножин з умовою мінімальної відмінності елементів

Завдання полягає у генеруванні всіх підмножин множини  $A = \{a_1, a_2, a_3\}$  з умовою мінімальної відмінності сусідніх породжуваних підмножин.

Для розв'язання цієї задачі необхідно поставити у відповідність кожному елементу множини  $a_i$  той же по номеру розряд числа у коді Грея. Формування чергової підмножини множини  $A$  відбувається шляхом виключення з множини  $A$  тих елементів, для яких відповідні розряди коду Грея дорівнюють 0.

Результати представимо у вигляді таблиці:

$i$	$b_1 b_2 b_3$	$g_1 g_2 g_3$	$B_i$
0	000	000	$\emptyset$
1	001	001	$a_3$

$i$	$b_1 b_2 b_3$	$g_1 g_2 g_3$	$B_i$
2	010	011	$a_2, a_3$
3	011	010	$a_2$
4	100	110	$a_1, a_2$
5	101	111	$a_1, a_2, a_3$
6	110	101	$a_1, a_3$
7	111	100	$a_1$

Для реалізації цього алгоритму використаємо перший спосіб генерації коду Грея.

### 5.2.12. Блок-схема першого алгоритму генерації підмножин з умовою мінімальної відмінності елементів

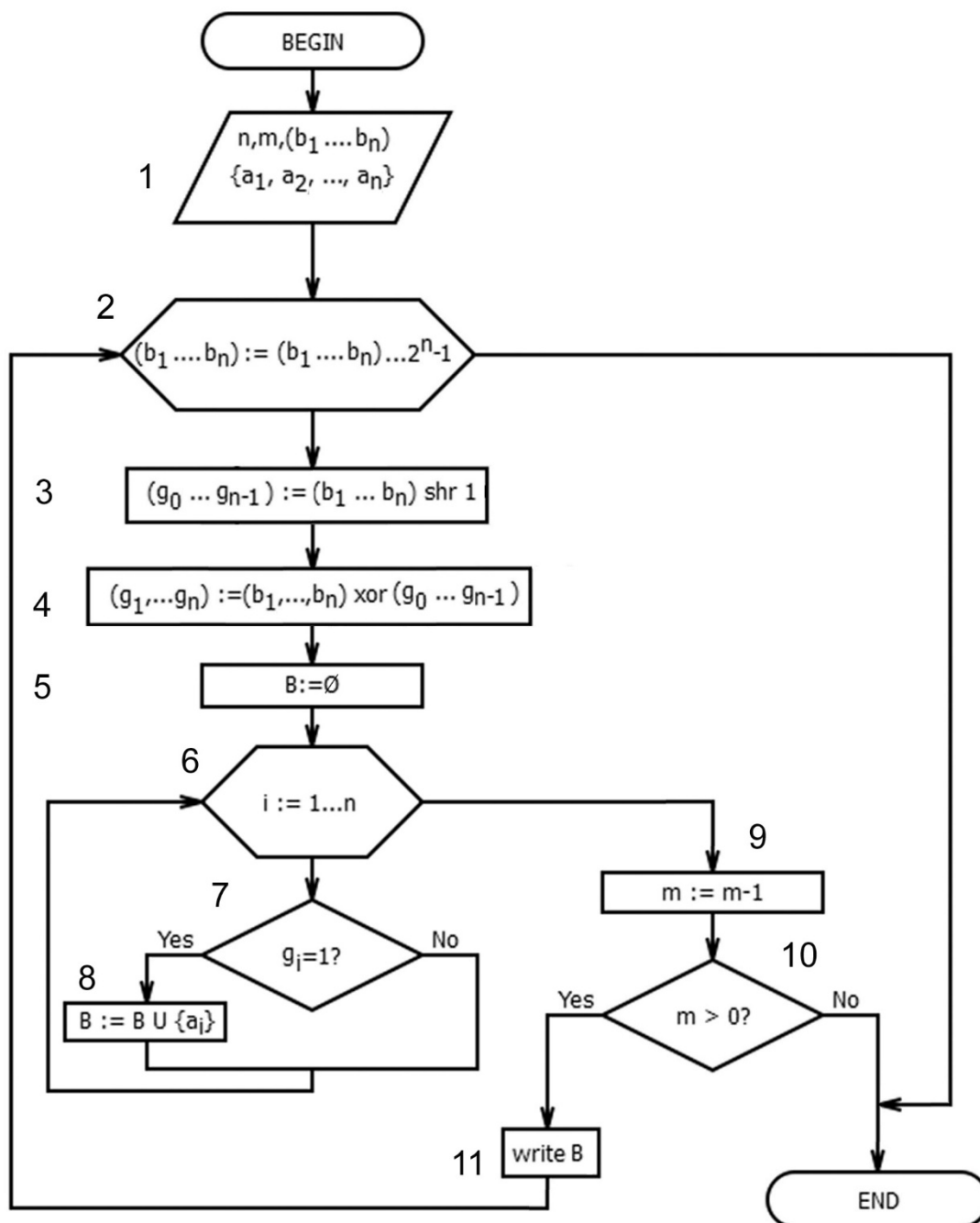


Рис. 5.6. Блок-схема першого алгоритму генерації підмножин з умовою мінімальної відмінності елементів

**Блок 1.** Ввід початкових параметрів алгоритму.

$n$  – кількість розрядів числа у двійковій системі числення.

$m$  – кількість чисел у коді Грея, які потрібно згенерувати.

$(b_1 b_2 \dots b_n)$  – початкове число у двійковій системі числення.

$\{a_1, a_2, \dots, a_n\}$  – базова множина для формування підмножин з умовою мінімальної відмінності елементів.

**Блок 2.** Цикл генерації чисел у коді Грея, починаючи з числа  $(b_1 b_2 \dots b_n)$ .

**Блок 3.** Операція зсуву вправо на один розряд числа  $(b_1 \dots b_n)$ . Результатом зсуву є число  $(g_0 g_1 \dots g_{n-1})$ .

**Блок 4.** Операція порозрядного додавання по модулю два числа  $(b_1 \dots b_n)$  та числа  $(g_0 g_1 \dots g_{n-1})$ . Результатом цієї операції є число  $(g_1 g_2 \dots g_n)$ , яке є числом у коді Грея, що відповідає числу  $(b_1 b_2 \dots b_n)$  у двійковій системі числення.

**Блок 5.** Встановлення в початковий стан для формування чергової підмножини з використанням числа у коді Грея.

**Блок 6.** Цикл формування підмножини шляхом послідовного аналізу розрядів числа  $(g_1 g_2 \dots g_n)$  у коді Грея.

**Блок 7.** Перевірка значення розряду числа у коді Грея.

**Блок 8.** Операція виконується, якщо значення поточного розряду числа у коді Грея дорівнює 1. У цьому випадку до підмножини додається відповідний елемент  $a_i$  множини  $\{a_1, a_2, \dots, a_n\}$ .

**Блок 9.** Зміна значення лічильника кількості згенерованих підмножин.

**Блок 10.** Перевірка стану лічильника кількості згенерованих підмножин.

**Блок 11.** Роздрукування чергової підмножини  $B$ .

### 5.2.13. Другий алгоритм генерації підмножин з умовою мінімальної відмінності елементів

Як і в попередньому випадку, будемо генерувати підмножини множини  $X = \{x_1, x_2, x_3\}$  з умовою мінімальної відмінності сусідніх породжуваних підмножин.

Для цього поставимо у відповідність кожному елементу  $x_i$  множини  $X$  такий же за номером розряд числа  $(g_1, \dots, g_n)$  у коді Грея. Формування чергової підмножини множини  $X$  відбувається шляхом додавання до пустої множини  $B$  тих елементів множини  $X$ , для яких відповідні розряди числа у коді Грея дорівнюють 1.

Модифікуємо другий алгоритм генерації чисел у коді Грея.

#### 5.2.14. Блок-схема другого алгоритму генерації підмножин з умовою мінімальної відмінності елементів

**Блок 1.** Ввід початкових параметрів алгоритму.

$n$  – кількість розрядів чисел у коді Грея, які потрібно згенерувати.

$A = (00, 01, 11, 10)$  - початкова послідовність чисел, яка відповідає двохрановним числам у коді Грея.

$\{x_1, x_2, \dots, x_n\}$  - базова множина для формування підмножин з умовою мінімальної відмінності елементів.

**Блок 2.** Початкова установка лічильника поточної кількості розрядів чисел, що генеруються алгоритмом у коді Грея.

**Блок 3.** Лічильник кількості розрядів. Алгоритм починає свою роботу з початкової кількості розрядів, яка дорівнює 2.

**Блок 4.** Цикл генерації  $(i+1)$ -розрядної послідовності чисел у коді Грея на основі  $i$  – розрядної послідовності.

**Блок 5.** Зсув вліво на один розряд чисел початкової послідовності у коді Грея. Операція аналогічна множенню даного числа на 2.

Наприклад:  $0100 * 0010 = 1000$  та  $0100 \text{shl} 1 = 1000$

Ця операція дозволяє сформувати першу половину  $(i+1)$ –розрядної послідовності чисел у коді Грея.

**Блок 6.** До числа, сформованого у попередньому блоці, додаємо 1 в молодший розряд. Одержані числа записуємо, починаючи з кінця  $(i+1)$  – розрядної послідовності. Таким чином формуємо другу половину  $(i+1)$ –розрядної послідовності чисел у коді Грея.

**Блок 7.** Перевірка ознаки досягнення розрядності  $n$  послідовності чисел у коді Грея.

**Блок 8.** Цикл формування підмножини шляхом послідовного аналізу розрядів числа  $(g_1 g_2 \dots g_n)$  у коді Грея.

**Блок 9.** Перевірка значення розряду числа у коді Грея.

**Блок 10.** Операція виконується, якщо значення поточного розряду числа у коді Грея дорівнює 1. У цьому випадку до підмножини додається відповідний елемент  $x_k$  множини  $\{x_1, x_2, \dots, x_k, \dots, x_n\}$ .

**Блок 11.** Роздруківка послідовності  $n$ -розрядних чисел у коді Грея.

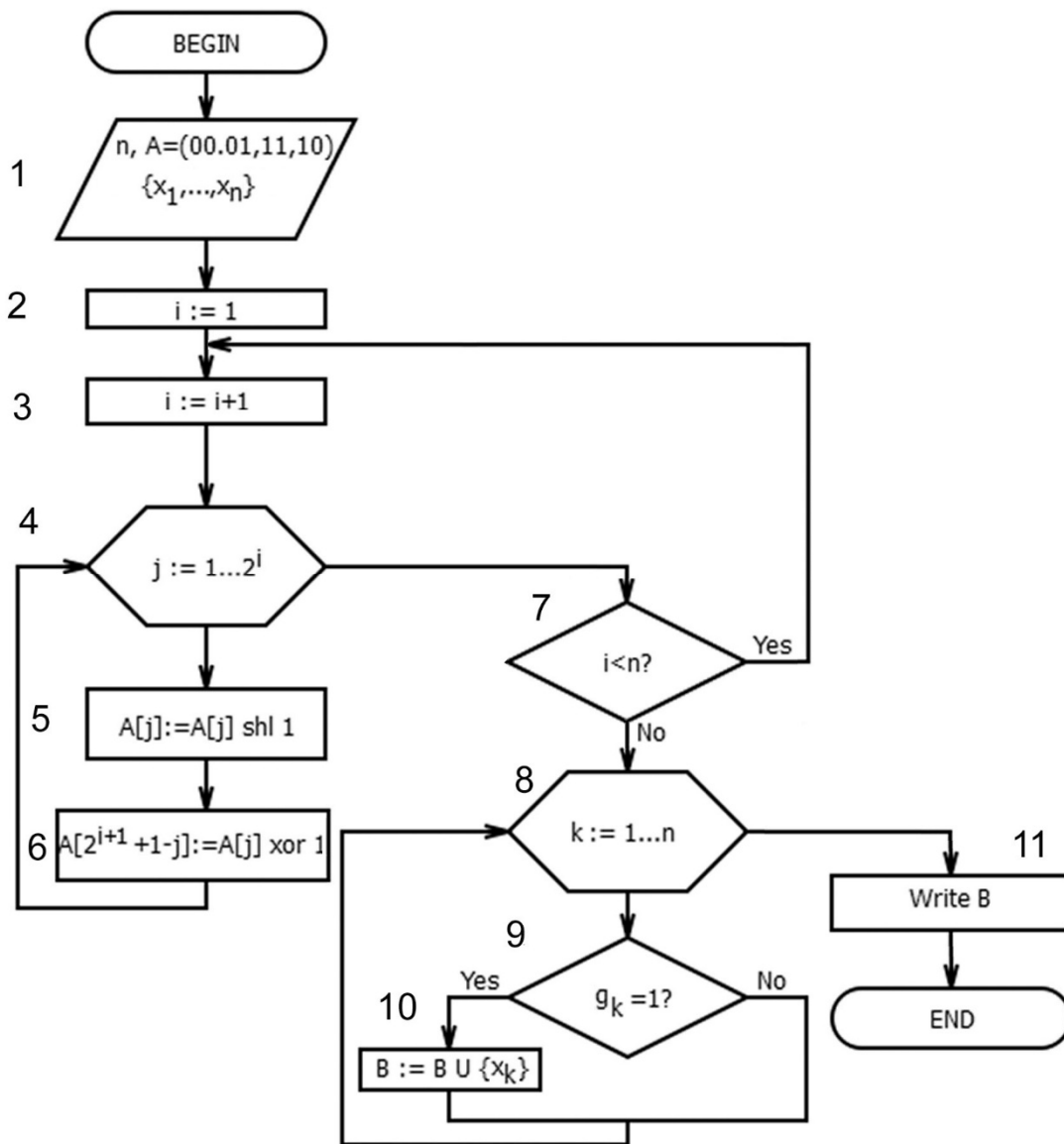


Рис. 5.7. Блок-схема другого алгоритму генерації підмножин з умовою мінімальної відмінності елементів

**5.2.15. Алгоритм генерації сполучень з  $n$  по  $k$  в лексикографічному порядку**

Сполученням з  $n$  елементів по  $k$  називають неупорядковану вибірку  $k$  елементів із заданих  $n$  елементів. Ми будемо генерувати всі сполучення з  $n$  по  $k$  для заданої  $n$ -елементної множини  $A$ . Число сполучень з  $n$  по  $k$  дорівнює біноміальному коефіцієнту

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Тому найкраща складність, яку можна чекати для алгоритму генерації всіх сполучень, дорівнює  $O(C_n^k)$ . Без обмеження спільності можна припускати  $A = \{1, 2, \dots, n\}$ . Довільне сполучення з  $n$  по  $k$  зручно представити у вигляді послідовності довжини  $k$  з чисел, упорядкованих за зростанням зліва направо. Всі такі послідовності природно породжувати в лексикографічному порядку. Наприклад, при  $n = 5$  і  $k = 3$  послідовність всіх сполучень в лексикографічному порядку наступна:

123, 124, 125, 134, 135, 145, 234, 235, 245, 345.

Очевидно, що при генерації всіх можливих сполучень перший елемент у лексикографічному порядку є сполучення  $(1, 2, \dots, k)$ , а останній-  $(n - k + 1, n - k + 2, \dots, n - 1, n)$ .

1. Розглянемо сполучення  $(a_1, a_2, \dots, a_k)$ .

2. Тоді наступне сполучення визначають з виразу:

$$(b_1, b_2, \dots, b_k) = (a_1, \dots, a_{p-1}, a_p + 1, a_p + 2, \dots, a_p + k - p + 1), \text{ де}$$

$$p = \max \{i | a_i < n - k + 1\}$$

3. Наступне сполучення, яке генерується на основі сполучення  $(b_1, b_2, \dots, b_k)$ , має вигляд:

$$(c_1, \dots, c_k) = (b_1, \dots, b_{p'-1}, b_{p'} + 1, b_{p'} + 2, \dots, b_{p'} + k - p' + 1), \text{ де}$$

$$p' = \begin{cases} p - 1, & \text{при } b_k = n, \\ k, & \text{при } b_k < n \end{cases}$$

**Приклад.** Нехай дано початковий терм  $A = (a_1, a_2, a_3, a_4, a_5) = (1, 2, 3, 4, 5)$ . Знайти сполучення з  $n = 5$  по  $k = 3$  у лексикографічному порядку.

Тоді перший елемент:  $(a_1, a_2, a_3) = (1, 2, 3)$ .

Останній елемент:  $(a_3, a_4, a_5) = (3, 4, 5)$ .

Даний алгоритм дозволяє генерувати всі сполучення  $C_n^k$  у випадку, коли як початкове сполучення вибрано перший елемент  $(1, 2, 3, \dots, k)$ .

**Приклад.** Для послідовності  $(1, 2, 3, 4, 5)$  при обчисленні  $C_5^3$  початковим елементом є  $(1, 2, 3)$ .

Якщо як початкову послідовність вибрати  $(i + 1, i + 2, \dots, i + k)$ , то алгоритм забезпечує генерацію сполучень, починаючи з  $(i + 1, i + 2, \dots, i + k)$  і закінчуючи  $(n - k + 1, n - k + 2, \dots, n - 1, n)$ .

**Приклад.** Для послідовності  $(1, 2, 3, 4, 5)$  при обчисленні  $C_5^3$  початковим сполученням виберемо послідовність  $(2, 3, 5)$ . Тоді з повної послідовності  $(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)$



алгоритм забезпечить генерування таких сполучень: (2,3,5), (2,4,5), (3,4,5).

За необхідності кількість згенерованих сполучень може бути обмеженою.

**Приклад.** Для послідовності (1,2,3,4,5) при обчисленні  $C_5^3$  початковим сполученням виберемо послідовність (1,2,5) і поставимо вимогу генерації трьох сполучень.

Тоді з повної послідовності

(1,2,3), (1,2,4), (1,2,5), (1,3,4), (1,3,5), (1,4,5), (2,3,4), (2,3,5), (2,4,5), (3,4,5)

алгоритм забезпечить генерування таких сполучень: (1,2,5), (1,3,4), (1,3,5).

### 5.2.16. Блок-схема алгоритму генерації сполучень з $n$ по $k$ в лексикографічному порядку

**Блок 1.** Ввід початкових даних:

$n$  – довжина початкової послідовності.

$k$  – кількість елементів сполучення.

$r$  – кількість сполучень, які необхідно згенерувати.

$(a_1, a_2, \dots, a_k)$  – початкове сполучення, з якого алгоритм починає генерувати наступні сполучення у лексикографічному порядку.

$a_1 < a_2 < \dots < a_k$  – умова задавання початкового сполучення.

**Блок 2.** Установка в початковий стан лічильника кількості елементів початкового сполучення  $m := 1$ .

**Блок 3.** Перевірка ознаки закінчення оцінки умови правильності задавання елементів початкового сполучення.

**Блок 4.** Блок виконується у випадку, коли ще не всі елементи початкового сполучення перевірені. В цьому випадку порівнюється поточний елемент початкового сполучення з наступним елементом. Якщо  $a_m < a_{m+1}$ , то елемент  $a_m$  задано правильно.

**Блок 5.** Значення елемента початкового сполучення  $a_m$  записуються в  $m$ -й елемент масиву  $A[m] := a_m$ .

**Блок 6.** Інкремент лічильника кількості елементів початкового сполучення  $m := m + 1$ .

**Блок 7.** Якщо лічильник  $m$  досяг максимально допустимого значення, тобто  $m = k$ , записуємо цей останній елемент в масив:  $A[m] := a_m$ .

*Увага! Далі змінна  $m$  буде використовуватися в новому смислового значенні.*

*Тепер вона виконує роль ознаки закінчення генерації всіх можливих сполучень з  $n$  по  $k$ .*

**Блок 8.** Перевіряємо випадок, коли довжина сполучення дорівнює довжині початкової послідовності, тобто чи  $n = k$ .

**Блок 9.** Якщо  $n = k$ , то змінній  $m$  надаємо значення 1.

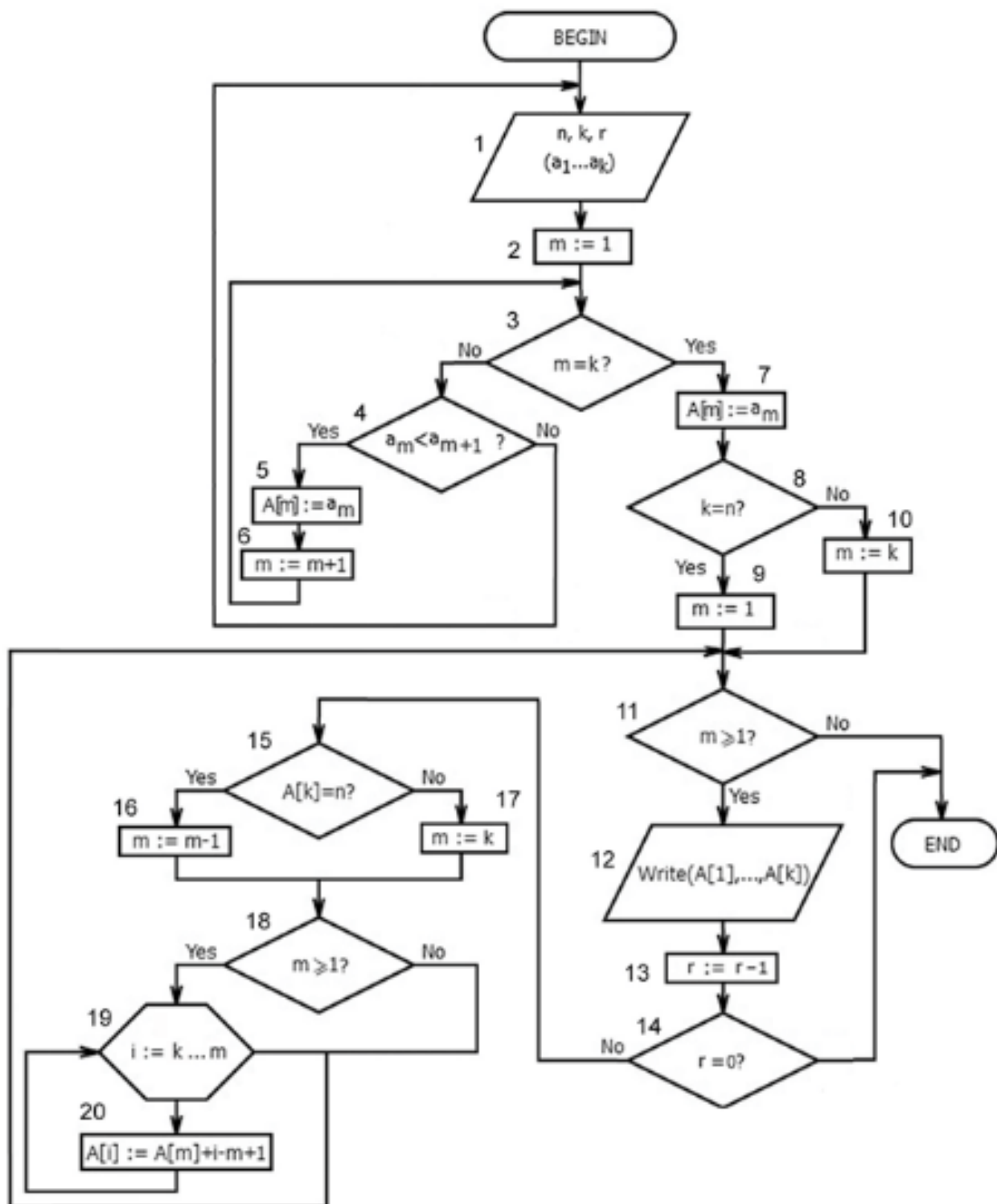


Рис. 5.8. Генерація сполучень з  $n$  по  $k$  в лексикографічному порядку

**Блок 10.** Якщо  $k < n$ , то  $m := k$ .

**Блок 11.** Перевіряємо ознаку закінчення роботи алгоритму  $m$ .

Якщо  $m \geq 1$ , то переходимо до роздруківки чергового сполучення. В протилежному випадку закінчуємо роботу алгоритму.

**Блок 12.** Роздруківка масиву поточного сполучення  $A$ .

**Блок 13.** Декремент лічильника кількості замовлених сполучень. Цей блок використовують у випадку, коли потрібно згенерувати наперед задану обмежену кількість сполучень. Якщо алгоритм повинен згенерувати всі можливі сполучення, починаючи з початкового, то задають  $r > C_n^k$ .

**Блок 14.** Перевірка ознаки кількості замовлених сполучень. Якщо  $r = 0$ , то всі замовлені сполучення згенеровано і алгоритм закінчує роботу. В протилежному випадку відбувається перехід до наступної частини алгоритму, яка забезпечує формування нового сполучення.

**Блоки 15-20.** Алгоритм генерації чергового сполучення.

#### 5.2.17. Алгоритм генерації сполучень з $n$ по $k$ на множині

Нехай дано довільну множину  $R = \{r_1, r_2, \dots, r_n\}$ . Необхідно створити алгоритм, який забезпечує побудову всіх сполучень з  $n$  по  $k$  на множині  $R$ . Для реалізації цього алгоритму модифікуємо попередній алгоритм генерації сполучень у лексикографічному порядку. Модифікація полягає в тому, що при формуванні сполучень будемо використовувати одержані числові сполучення як індекси сполучень елементів множини  $R$ .

**Приклад.** Розглянемо генерацію сполучень з 5 по 3 на множині  $R = \{r_1, r_2, r_3, r_4, r_5\}$ . Для цього сформуємо послідовність  $A = (1, 2, 3, 4, 5)$  та застосуємо до неї алгоритм генерації сполучень при  $n = 5$  та  $k = 3$ . В результаті одержимо сполучення:

$(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)$

Використавши ці сполучення як індекси елементів множини  $R$ , можемо записати сполучення з 5 по 3 даної множини:

$(r_1, r_2, r_3), (r_1, r_2, r_4), (r_1, r_2, r_5), (r_1, r_3, r_4), (r_1, r_3, r_5),$   
 $(r_1, r_4, r_5), (r_2, r_3, r_4), (r_2, r_3, r_5), (r_2, r_4, r_5), (r_3, r_4, r_5)$

#### 5.2.18. Блок-схема алгоритму генерації сполучень з $n$ по $k$ на множині

**Блок 1.** Ввід початкових даних:

$n$  – потужність множини  $R = \{r_1, r_2, \dots, r_n\}$ .

$k$  – кількість елементів сполучення.

$\{r_1, r_2, \dots, r_n\}$  – елементи множини  $R$ .

**Блоки 2, 3.** Генерація числової послідовності індексів елементів множини  $R$ .

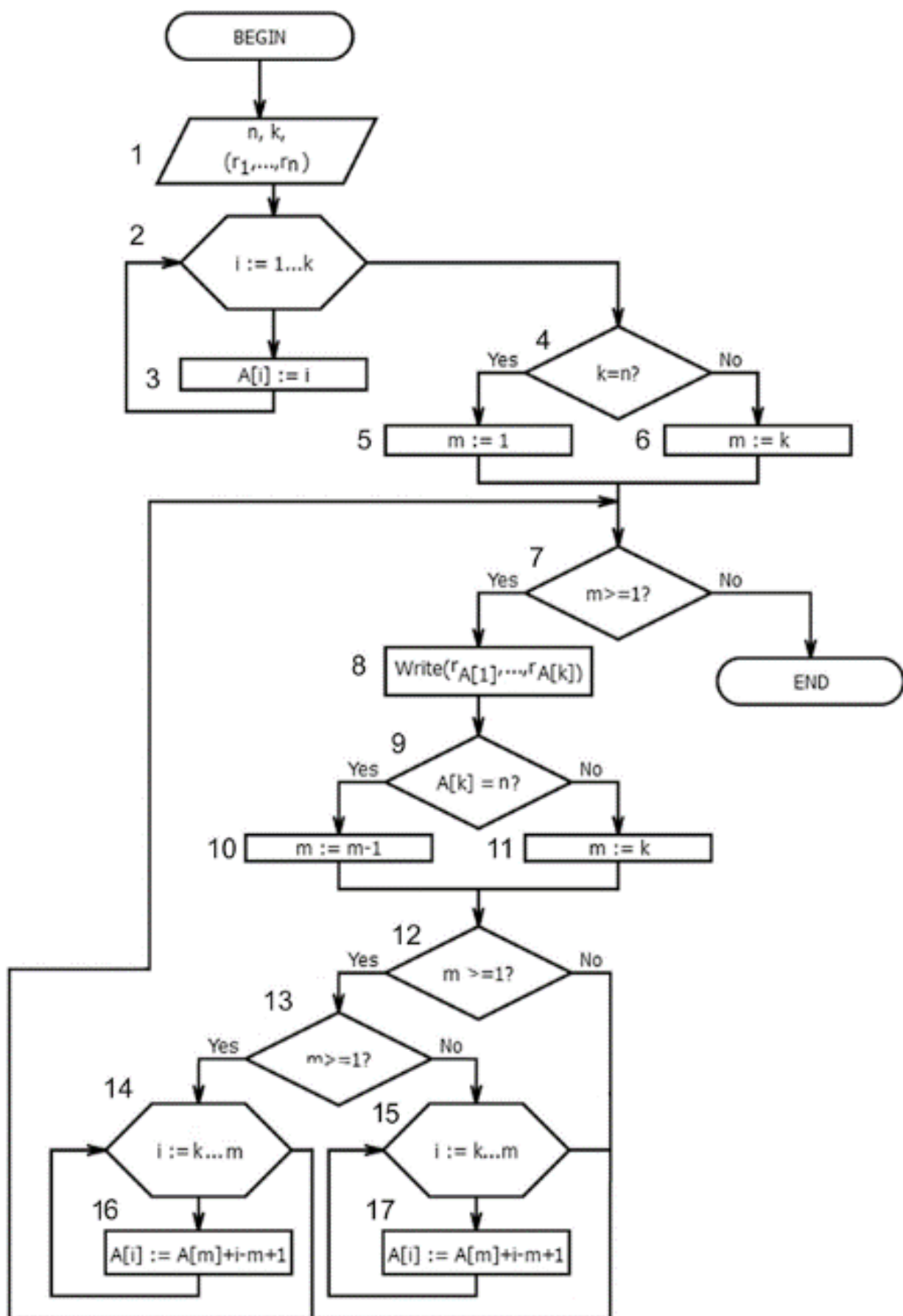


Рис. 5.9. Генерація сполучень з  $n$  по  $k$  на множині

**Блоки 4, 5, 6.** Встановлення початкового значення змінної  $m$ , нульове значення якої є ознакою закінчення роботи алгоритму.

**Блок 7.** Перевірка ознаки закінчення роботи алгоритму. Якщо  $m \geq 1$ , то алгоритм продовжує свою роботу. В протилежному випадку, тобто при  $m < 1$ , алгоритм зупиняється.

**Блок 8.** Роздруківка чергового сполучення елементів множини  $R$  за умови, що індекси цих елементів дорівнюють відповідним значенням елементів масиву  $A$ .

**Блок 9.** Перевіряємо, чи не містить останній елемент сполучення  $A[k]$  максимального значення  $n$ .

**Блок 10.** Якщо  $A[k] = n$ , то виконуємо декремент змінної  $m$ :  $m := m - 1$

**Блок 11.** Якщо  $A[k] \neq n$ , то змінну  $m$  встановлюємо  $m := k$ .

**Блок 12.** Перевірка ознаки закінчення роботи алгоритму. Якщо  $m \geq 1$ , то відбувається генерація чергового сполучення. В протилежному випадку, тобто при  $m < 1$ , відбувається перехід на початок головного циклу та завершення роботи алгоритму.

**Блок 13, 14.** Цикл генерації чергового сполучення з числових індексів, записаних в масиві  $A$ .

### 5.2.19. Алгоритм генерації розбиття числа $n$ у словниковому порядку

На множині цілочисельних послідовностей визначимо словниковий порядок  $\leq$  (порівняйте визначення словникового порядку з лексикографічним). Нехай  $a = (a_1, \dots, a_q)$  і  $b = (b_1, \dots, b_s)$  – довільні цілочисельні послідовності, можливо, різної довжини.

**Визначення.**  $(a_1, \dots, a_q) \leq (b_1, \dots, b_s)$ , якщо виконується хоча б одна з умов:

або  $q \leq s$  і  $a_i = b_i$  для будь-якого  $i \leq q$ , або існує  $p \leq \min(s, q)$  таке, що  $a_p < b_p$  і  $a_i = b_i$  для всіх  $i < p$ . Бінарне відношення  $\leq$  є лінійним порядком. Для розбиттів числа  $n$  визначення словникового порядку дещо спрощується.

**Зауваження.** Якщо  $a, b$  – розбиття числа  $n$ , то

$$a < b \Leftrightarrow \exists p \leq \min(s, q) (a_p < b_p \ \& \ \forall i < p (a_i = b_i))$$

Розбиття числа  $n$  будемо породжувати в словниковому порядку. Ясно, що першою розбивкою (найменшим елементом) в словниковому порядку буде послідовність  $(1, 1, \dots, 1)$  довжини  $n$ , а останньою (найбільшим елементом) – одноелементна послідовність  $(n)$ . З'ясуємо вид розбиття, яке є безпосередньо наступним за розбиттям  $a = (a_1, \dots, a_q)$  в словниковому порядку. Знайдемо таку найбільш праву позицію  $p < q$ , в якій число  $a_p$  можна збільшити на 1, зберігши

властивість спадання послідовності. Далі суму доданків, що залишилися, за

мінусом одиниці  $\left( \sum_{i=p+1}^q a_i - 1 \right)$ , представимо у вигляді суми одиниць.

Неважко довести, що саме таке розбиття  $b$  безпосередньо слідує за  $a$ . При переході від  $a$  до  $b$  число необхідних змін над послідовністю  $a$  є величина змінна, залежна від  $n$ . Цю залежність можна усунути, якщо перейти до іншого способу задавання розбиття  $a = (a_1, \dots, a_q)$ . Впорядкуємо всі різні числа  $(a_{i_1}, \dots, a_{i_k})$

серед  $(a_1, \dots, a_q)$  у порядку зменшення  $a_{i_1} > \dots > a_{i_k}$ .

Нехай  $m_j$  – число входжень  $a_{i_j}$  в розбиття  $a$  і  $m = (m_1, \dots, m_k)$ . Ясно, що розбиття  $a$  числа  $n$  однозначно визначається парою послідовностей  $(a_{i_1}, \dots, a_{i_k})$  і  $(m_1, \dots, m_k)$ .

Тому надалі будь-яке розбиття  $a$  числа  $n$  будемо записувати у вигляді  $a = (a_1 \cdot m_1, \dots, a_k \cdot m_k)$ ,

$$\text{де } a_1 > a_2 > \dots > a_k > 0, m_i > 0, i = 1, 2, \dots, k, 1 \leq k \leq n, m = \sum_{i=1}^k m_i a_i.$$

Елемент  $m_i \cdot a_i$  представляє собою послідовність  $a_i, a_i, a_i, \dots, a_i$  довжини  $m_i$  і називається блоком розбиття. Очевидно, що розбиття  $a$  є найменшим при  $k = 1$  і  $m_k = n$ , а найбільшим при  $k = m_k = 1$ . Таке представлення розбиття числа  $n$  виключає необхідність пошуку позиції  $p$  при перегляді справа наліво поточного розбиття.

Покажемо, що для перестроювання розбиття  $a$  в безпосередньо наступне розбиття  $b$  потрібно змінити не більше двох блоків.

**Теорема.** Нехай  $a = (a_1 \cdot m_1, \dots, a_k \cdot m_k)$  – розбиття числа  $n$  і розбиття  $b$  безпосередньо слідує за  $a$  в словниковому порядку. Тоді

1. Якщо  $m_k = 1$ , то  $k \geq 2$  і

$$b = (m_1 \cdot a_1, \dots, m_{k-2} \cdot a_{k-2}, 1 \cdot (a_{k-1} + 1), S' \cdot 1).$$

2. Якщо  $m_k \geq 2, k \geq 2$  і  $a_{k-1} = a_k + 1$ , то

$$b = (m_1 \cdot a_1, \dots, m_{k-2}, a_{k-2}, (m_{k-1} + 1) \cdot a_{k-1}, S \cdot 1).$$

3. Якщо  $m_k \geq 2, k \geq 2$  і  $a_{k-1} \neq a_k + 1$ , то

$$b = (m_1 \cdot a_1, \dots, m_{k-1}, a_{k-1}, 1 \cdot (a_k + 1), S \cdot 1).$$

4. Якщо  $k = 1$  то  $b = (1 \cdot (a_k + 1), S \cdot 1)$ .

$$\text{Тут } S' = m_k a_k + m_{k-1} a_{k-1} - (a_{k-1} + 1) ; S = m_k a_k - (a_k + 1).$$

**Доведення.** При переході від  $a$  до  $b$  необхідно найбільш правий можливий елемент розбиття  $a$  збільшити на 1. Такий елемент  $x$  буде першим елементом блоку, до якого він входить. Розглянемо два можливі випадки.

**Випадок 1.**  $m_k = 1$ . Елемент  $a_k$  можна збільшити, зберігши розбиття числа  $n$ . Оскільки  $a$  - не найбільше розбиття, маємо  $k \geq 2$ . Оскільки  $a_{k-1} + a_k \geq a_k + 1$ , то  $x = a_{k-1}$  і  $x$  є першим елементом  $(k-1)$  блоку. Цей елемент збільшуємо на 1, а суму доданків, що залишилися,  $S' = m_k a_k + m_{k-1} a_{k-1} - (a_{k-1} + 1)$ , представляємо у вигляді суми одиниць.

**Випадок 2.**  $m_k \geq 2$ . Тоді  $m_k a_k \geq a_k + 1$ . Отже,  $x = a_k$  і  $x$  є першим елементом  $k$ -го блоку. Цей елемент збільшуємо на 1, а суму доданків, що залишилися,  $S = m_k a_k - (a_k + 1)$  представляємо у вигляді суми одиниць. У кожному з цих випадків залишається перерахувати значення  $a_i$  і  $m_i$  для не більше, ніж двох змінених блоків. Теорема доведена.

**Зауваження.** Якщо вибрати значення  $a_0$  таке, що  $a_0 \neq a_1 + 1$ , то в наведеній теоремі можна виключити (4) та умову  $k \geq 2$  в (2), (3), зберігши справедливим твердження теорема.

**Приклад.** Розбиття числа 7 в словниковому порядку:

$$(7 \cdot 1) = (1, 1, 1, 1, 1, 1, 1),$$

$$(1 \cdot 2, 5 \cdot 1) = (2, 1, 1, 1, 1, 1),$$

$$(2 \cdot 2, 3 \cdot 1) = (2, 2, 1, 1, 1),$$

$$(3 \cdot 2, 1 \cdot 1) = (2, 2, 2, 1),$$

$$(1 \cdot 3, 4 \cdot 1) = (3, 1, 1, 1, 1),$$

$$(1 \cdot 3, 1 \cdot 2, 2 \cdot 1) = (3, 2, 1, 1),$$

$$(1 \cdot 3, 2 \cdot 2) = (3, 2, 2),$$

$$(2 \cdot 3, 1 \cdot 1) = (3, 3, 1),$$

$$(1 \cdot 4, 3 \cdot 1) = (4, 1, 1, 1),$$

$$(1 \cdot 4, 1 \cdot 2, 1 \cdot 1) = (4, 2, 1),$$

$$(1 \cdot 4, 1 \cdot 3) = (4, 3),$$

$$(1 \cdot 5, 2 \cdot 1) = (5, 1, 1),$$

$$(1 \cdot 5, 1 \cdot 2) = (5, 2),$$

$$(1 \cdot 6, 1 \cdot 1) = (6, 1),$$



$$(1 \cdot 7) = (7).$$

Розглянемо блок-схему цього алгоритму.

**5.2.20. Блок-схема алгоритму генерації розбиття числа  $n$  у словниковому порядку**

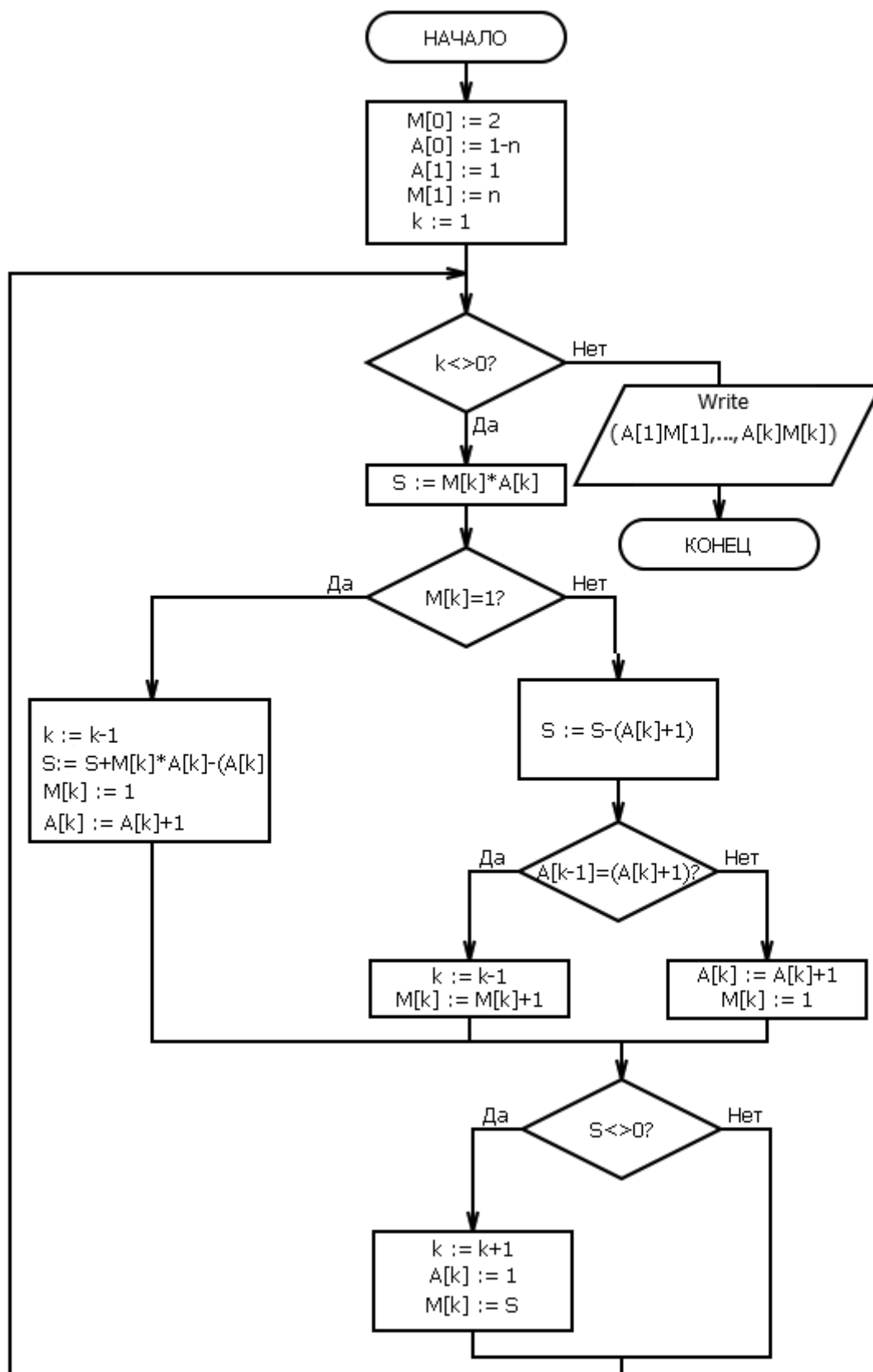


Рис. 5.10. Генерація розбиття числа  $n$  у словниковому порядку

### 5.2.21. Швидке сортування QuickSort

#### Псевдокод процедури швидкого сортування:

Процедура QuickSort(A: масив, g – початковий індекс, r – кінцевий індекс);

#### Початок процедури

1. Вибрати **supp** - елемент з середнім індексом (опорний):
2. Почати перегляд від початку масиву та знайти елемент, що є більшим за опорний  $A[i] > x$
3. Почати перегляд з кінця масиву та знайти елемент, що є меншим за опорний  $A[j] < x$
4. Якщо попередні процеси ще не перетнулися ( $i$  не більше  $j$ ), то
  - 4.1. Поміняти знайдені елементи місцями
  - 4.2. Перейти до п. 2., але не з початку масиву, а з місця попередньої зупинки
5. Проаналізувати індекси останнього обміну.
  - 5.1 Якщо  $i$  менше  $r$ , то запустити QuickSort(A, i, r)
  - 5.2 Якщо  $j$  більше  $g$ , то запустити QuickSort(A, g, j)

#### Кінець процедури

#### Опис програми швидкого сортування

У наведеному алгоритмі використані наступні позначення:

$a[k]$  - масив чисел, у якому проводиться сортування.

Процедура дозволяє сортувати довільну підмножину масиву  $a[k]$

$g$  – номер мінімального елемента масиву, з якого починається сортування.

$r$  – номер максимального елемента масиву, на якому закінчується сортування.

Суть методу

1. На кожній ітерації виділяють частину масиву чисел — робочий масив.

На першій ітерації – це весь масив чисел  $a[k]$ , у якому проводиться сортування.

Встановлюємо границі робочого масиву  $g = 1$  і  $r = n$ .

2. Вибираємо елемент  $x := a[(g+r) \div 2]$ , який розміщений посередині робочого масиву.

3. Далі, починаючи з  $i = 1$ , послідовно збільшуємо значення  $i$  на одиницю й порівнюємо кожний елемент  $a_i$  з  $x$ , поки не буде знайдено елемент  $a_i$  такий, що  $a_i > x$ .

4. Потім, починаючи з  $j = r$ , послідовно зменшуємо значення  $j$  на одиницю, поки не буде знайдений елемент  $a_j$  такий, що  $x > a_j$ .

5. Якщо для знайдених елементів  $a_i$  і  $a_j$  виконується умова  $i \leq j$ , то ці елементи в робочому масиві міняються місцями.

6. Описана процедура закінчиться знаходженням головного елемента й поділом масиву на дві частини: одна частина буде містити елементи, менші за  $x$ , а інша — більші за  $x$ .

$\leq x$	$x$	$x \geq$
----------	-----	----------

Далі для кожної такої частини знову застосовується описана вище процедура. Паскаль-Програма, що реалізує даний алгоритм для 10-елементного масиву, має такий вигляд:

```

Program Qsort;
Const N=10;
Var a:array[1..N] of integer; (* вхідний масив *)
    k:integer;
procedure Quicksort(g,r:integer);
(* Процедура швидкого сортування*)
var i,j,x,y: integer;
begin
    i:=g; j:=r; x:= a[(g+r) div 2];
    repeat
        while (a[i]<x) do inc(i);
        while (x<a[j]) do dec(j);
        if (i<=j) then
            begin
                y:=a[i]; a[i]:=a[j]; a[j]:=y; inc(i); dec(j);
            end;
    until (i>j);
(*Рекурсивне використання процедури Quicksort *)
if (g<j) then Quicksort(g,j);
if (i<r) then Quicksort(i,r);
end;
begin
    writeln('Уведіть',N, 'елементів масиву:') ;
    for k:=1 to N do readln(a[k]);
    Quicksort(1,N);
    writeln('Після сортування:');
    for до:=1 to N do write(a[k], ' ');
end.

```

### 5.2.22. Сортування Шелла

Сортування Шелла отримало свою назву по імені її творця Д. Л. Шелла. Однак, цю назву можна вважати вдалою, оскільки виконувані при сортуванні дії нагадують вкладання морських черепашок одна на одну.

Загальний метод, який використовує сортування вставкою, застосовує принцип зменшення відстані між порівнюваними елементами.

На рисунку показана схема виконання сортування Шелла для масиву

{f, d, a, c, b, e}. Спочатку сортуються всі елементи, які зміщені один від одного на три позиції. Потім сортуються всі елементи, які зміщені на дві позиції. І, нарешті, упорядковуються всі сусідні елементи.

прохід 1	f	d	a	c	b	e
прохід 2	c	b	a	f	d	e
прохід 3	a	b	c	e	d	f
результат	a	b	c	d	e	f

### Ілюстрація сортування Шелла

```
{ Процедура сортування Шелла }
procedure Shell(var item: DataArray; count:integer);
const
    t = 5;
var
    i, j, k, s, m: integer;
    h: array[1..t] of integer;
    x: DataItem;
begin
    h[1]:=9; h[2]:=5; h[3]:=3; h[4]:=2; h[5]:=1;
    for m := 1 to t do
        begin

            k:=h[m];
            s:=-k;
            for i := k+1 to count do
                begin
                    x := item[i];
                    j := i-k;
                    if s=0 then
                        begin
                            s := -k;
                            s := s+1;
                            item[s] := x;
                        end;
                    while (x<item[j]) and (j<count) do
                        begin
                            item[j+k] := item[j];
                            j := j-k;
                        end;
                    item[j+k] := x;
                end;
            end;
        end; { кінець сортування Шелла }
```

Ефективність цього алгоритму пояснюється тим, що при кожному проході використовується відносно невелике число елементів або елементи масиву вже знаходяться у відносному порядку, а впорядкованість збільшується при кожному новому перегляді даних.

Відстані між порівнюваними елементами можуть змінюватися по-різному. Обов'язковим є лише те, що останній крок повинен дорівнювати одиниці. Наприклад, хороші результати дає послідовність кроків 9, 5, 3, 2, 1, яка використана в показаному вище прикладі. Слід уникати послідовностей зі степенем двійки, які, як показують складні математичні викладки, знижують ефективність алгоритму сортування. (Однак, при використанні таких послідовностей кроків між порівнюваними елементами це сортування буде, як і раніше, працювати правильно).

Внутрішній цикл має дві умови перевірки. Умова " $x < \text{item}[j]$ " необхідна для впорядкування елементів. Умови " $j > 0$ " і " $j \leq \text{count}$ " необхідні для того, щоб запобігти виходу за межі масиву " $\text{item}$ ". Ця додаткова перевірка деякою мірою погіршує сортування Шелла. Злегка змінені версії сортування Шелла використовують спеціальні керуючі елементи, які не є в дійсності частиною тієї інформації, яка повинна сортуватися. Керуючі елементи мають граничні для масиву даних значення, тобто найменше та найбільше значення. У цьому випадку не обов'язково виконувати перевірку на граничні значення. Однак, застосування таких керуючих елементів вимагає спеціальних знань про ту інформацію, яка сортується, і це знижує універсальність процедури сортування.

Аналіз сортування Шелла потребує вирішення деяких складних математичних задач. Час виконання сортування Шелла пропорційний  $n^{1.2}$ . Ця залежність значно краща від квадратичної залежності, якій підпорядковуються основні алгоритми сортування.

### 5.2.23. Методи пошуку

Пошук інформації в відсортованому масиві вимагає проведення послідовного перегляду масиву. Перегляд починається з першого елемента і завершується або знайденим елементом, або досягненням кінця масиву. Цей метод повинен використовуватися для невідсортованих даних, але він також може використовуватися для відсортованих даних. Якщо дані відсортовані, то може використовуватися двійковий пошук, який виконується значно швидше.

#### Послідовний пошук

Алгоритм послідовного пошуку має дуже простий вигляд. Нижче представлена функція, яка виконує пошук елемента із заданим значенням ключа в символьному масиві заданої довжини:

```

function SeqSearch(item:   DataArray;   count:integer;
key:DataItem):integer;
    var
        t:integer;
    begin
        t:=1;
        while (key<>item[t]) and (t<=count) t:=t+1;
        if t>count then SeqSearch:=0
        else SeqSearch:=t;
    end; { кінець послідовного пошуку }

```

Ця функція видає або значення індексу для знайденого елемента масиву, або нульове значення, коли необхідний елемент не знайдений. При прямому послідовному пошуку в середньому перевіряють  $n/2$  елементів. У кращому випадку буде перевірятися тільки один елемент, а в гіршому випадку будуть перевірятися  $n$  елементів. Якщо інформація розміщується на диску, то пошук може бути дуже тривалим. Однак, якщо дані не відсортовані, то послідовний пошук є єдиним можливим в даному випадку методом пошуку.

## Двійковий пошук

Якщо дані відсортовані, то може використовуватися дуже хороший метод пошуку, названий двійковим пошуком. При такому пошуку використовується метод "розділяй і володарюй". Спочатку проводиться перевірка середнього елемента. Якщо його ключ більший за ключ необхідного елемента, то здійснюють перевірку середнього елемента з першої половини. В іншому випадку здійснюють перевірку середнього елемента з другої половини. Цей процес повторюється доти, поки або не буде знайдений необхідний елемент, або не залишиться елементів для перевірки.

Наприклад, для пошуку числа 4 в масиві 1 2 3 4 5 6 7 8 9 зазначеним методом спочатку перевіряють середній елемент, яким є число 5. Оскільки цей елемент більше 4, пошук буде продовжений в першій половині масиву, тобто серед чисел 1 2 3 4 5. Тут середнім елементом є 3. Це значення менше 4 і тому перша половина не буде більше розглядатися і пошук триває серед чисел 4 5. На наступному кроці потрібний елемент буде знайдений. При двійковому пошуку число порівнянь в гіршому випадку  $O(\log n)$ . Для середнього випадку це значення буде дещо кращим, а в кращому випадку воно дорівнює одиниці.

Приведену нижче функцію, яка реалізує двійковий пошук для символьних масивів, можна використовувати для пошуку будь-якої довільної структури даних, змінивши блок порівняння і визначення типу даного "DataItem".

```

function BSearch (item:   DataArray;   count:integer;
key:DataItem):integer;
    var
        low, high, mid: integer;

```

```

    found:boolean;
begin
    low:=1; high:=count;
    found:=false;           { не знайдено }
    while (low<=high) and (not found) do
    begin
        mid:=(low+high) div 2;
        if key<item[mid] then high:=mid-1
        else if key>item[mid] then low:=mid+1
        else found:=true;   { знайдено }
    end;
    if found then BSearch:=mid
    else BSearch:=0;       { не знайдено }
end; { кінець пошуку }

```

### 5.3. ЗАВДАННЯ ЛАБОРАТОРНОЇ РОБОТИ

1. Вивчити принципи роботи алгоритму перестановок у лексикографічному порядку за блок-схемою, представленою на рис. 5.1.
2. Вивчити принципи роботи алгоритму генерації двійкових векторів довжини  $n$ , спираючись на блок-схему відповідного алгоритму, представленого на рис. 5.2.
3. Вивчити алгоритм генерації підмножин заданої множини, який використовує принципи роботи алгоритму генерації двійкових векторів, використовуючи блок-схему даного алгоритму, представлену на рис. 5.3.
4. Вивчити правила формування чисел у коді Грея та розглянути перший алгоритм генерації чисел у цьому коді, який використовує блок-схему першого алгоритму, представлену на рис. 5.4.
5. Вивчити правила формування чисел у коді Грея за другим алгоритмом генерації чисел у цьому коді, який використовує блок-схему другого алгоритму, представлену на рис. 5.5.
6. Використовуючи перший алгоритм генерації чисел у коді Грея, вивчити спосіб формування підмножин з умовою мінімальної відмінності елементів, використовуючи блок-схему першого алгоритму генерації підмножин з умовою мінімальної відмінності елементів, представлену на рис. 5.6.
7. Використовуючи другий алгоритм генерації чисел у коді Грея, вивчити спосіб формування підмножин з умовою мінімальної відмінності елементів за блок-схемою другого алгоритму генерації підмножин з умовою мінімальної відмінності елементів, представленою на рис. 5.7.
8. Вивчити принципи роботи алгоритму генерації сполучень з  $n$  по  $k$  в лексикографічному порядку, використовуючи блок-схему, представлену на рис. 5.8.
9. Вивчити принципи роботи алгоритму генерації сполучень з  $n$  по  $k$  на множині, використовуючи блок-схему, представлену на рис. 5.9.



10. Вивчити теоретичні основи та базові принципи роботи алгоритму генерації розбиття числа  $n$  у словниковому порядку, використовуючи блок-схему, представлену на рис. 5.10.
11. Вивчити принцип роботи алгоритму швидкого сортування QuickSort, використовуючи опис алгоритму та його псевдокод, що наведені в підрозділі 5.2.21.
12. Вивчити принцип роботи алгоритму сортування Шелла, використовуючи опис алгоритму, що наведений в підрозділі 5.2.22.
13. Вивчити принцип роботи алгоритмів пошуку, використовуючи опис алгоритмів, що наведені в підрозділі 5.2.23.

#### **5.4. Вимоги до програмного забезпечення:**

1. Лабораторна робота виконується з використанням скриптової мови програмування Python.
2. Для написання коду застосувати IDE PyCharm 3 Edu.
3. Для написання GUI застосувати бібліотеку tkinter.
4. Необхідно забезпечити ввід даних з клавіатури та з файлу.

#### **Зміст звіту:**

1. Титульний лист повинен мати такий вигляд:

Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»  
Факультет інформатики та обчислювальної техніки  
Кафедра обчислювальної техніки

Дискретна математика  
Лабораторна робота №5  
«Комбінаторика: перестановки, розміщення, сполучення»

Виконав:  
студент групи ІО-ХХ  
Прізвище, ім'я.  
Залікова книжка № \_\_\_\_\_  
Перевірів Новотарський М.А.

Київ 2016р.

2. Мета лабораторної роботи та загальне завдання
3. Короткі теоретичні відомості по темі.
4. Опис структури даних та принципів роботи комбінаторного алгоритму відповідно до варіанту.
5. Блок-схема, яка відповідає алгоритму, що використаний в лабораторній роботі.
6. Роздруківка того фрагменту тексту програми, який написаний індивідуально.
7. Роздруківка результатів виконання програми з контрольним прикладом.
8. Аналіз результатів та висновки.

### Контрольні запитання

1. Дати означення перестановок та вивести формулу визначення кількості (числа) перестановок.
2. Розміщення. Навести приклад розміщення.
3. Розміщення з повтореннями.
4. Сполучення.
5. Дати означення лексикографічного порядку
6. Дати означення словникового порядку.
7. Алгоритм перестановок у лексикографічному порядку.
8. Алгоритм генерації сполучень з  $n$  по  $k$  у лексикографічному порядку.
9. Алгоритм генерації розбиття числа  $n$  у словниковому порядку.

### 5.5. Варіанти для виконання лабораторної роботи

Номер варіанту  $I$  визначають як результат операції  $I = NZK \bmod 26+1$ , де  $NZK$  – номер залікової книжки. Номер варіанту відповідає номеру пункту завдання до лабораторної роботи.

№	Опис варіанта
1	Вивчити принципи роботи алгоритму перестановок у лексикографічному порядку. Написати програму перестановок чисел десяткової системи числення у лексикографічному порядку. Вхідні параметри мають такі значення: 1. Максимальне значення $n$ дорівнює $(10+NZK \bmod 10)$ . 2. Значення $s$ може змінюватися довільно від 1 до $n$ 3. Сформувати початкову перестановку $P$ таким чином, щоб кожен її елемент вибирався випадково та без повторень чисел.
2	Вивчити принципи роботи алгоритму перестановок у антилексикографічному порядку. Написати програму перестановок чисел десяткової системи числення у антилексикографічному порядку. Вхідні параметри мають такі значення: 1. Максимальне значення $n$ дорівнює $(10+NZK \bmod 11)$ . 2. Значення $s$ може змінюватися довільно від 1 до $n$ 3. Сформувати початкову перестановку $P$ таким чином, щоб кожен її елемент вибирався випадково та без повторень чисел.
3	Вивчити принципи роботи алгоритму перестановок у лексикографічному порядку. Написати програму перестановок букв латинського алфавіту у лексикографічному порядку. Вхідні параметри мають такі значення: 1. Максимальна кількість букв дорівнює $(10+NZK \bmod 15)$ . 2. Значення $s$ може змінюватися довільно від 1 до $n$ 3. Сформувати початкову перестановку $P$ таким чином, щоб кожен її елемент вибирався випадково та без повторень букв.
4	Вивчити принципи роботи алгоритму перестановок у антилексикографічному порядку. Написати програму перестановок букв латинського алфавіту у антилексикографічному порядку.

№	Опис варіанта
	<p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Максимальна кількість букв дорівнює <math>10 + NZK \bmod 16</math>.</li> <li>2. Значення <math>s</math> може змінюватися довільно від 1 до <math>n</math>.</li> <li>3. Сформувати початкову перестановку <math>P</math> таким чином, щоб кожен її елемент вибирався випадково та без повторень букв.</li> </ol>
5	<p>Вивчити принципи роботи алгоритму перестановок у лексикографічному порядку. Написати програму перестановок букв українського алфавіту у лексикографічному порядку.</p> <p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Максимальна кількість букв дорівнює <math>10 + NZK \bmod 17</math>.</li> <li>2. Значення <math>s</math> може змінюватися довільно від 1 до <math>n</math>.</li> <li>3. Сформувати початкову перестановку <math>P</math> таким чином, щоб кожен її елемент вибирався випадково та без повторень букв.</li> </ol>
6	<p>Вивчити принципи роботи алгоритму перестановок у антилексикографічному порядку. Написати програму перестановок букв українського алфавіту у антилексикографічному порядку.</p> <p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Максимальна кількість букв дорівнює <math>10 + NZK \bmod 18</math>.</li> <li>2. Значення <math>s</math> може змінюватися довільно від 1 до <math>n</math>.</li> <li>3. Сформувати початкову перестановку <math>P</math> таким чином, щоб кожен її елемент вибирався випадково та без повторень букв.</li> </ol>
7	<p>Вивчити принципи роботи алгоритму генерації двійкових векторів довжини <math>n</math>. Написати програму генерації двійкових векторів довжини <math>n</math> у лексикографічному порядку.</p> <p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Максимальне значення <math>n</math> дорівнює номеру залікової книжки (NZK).</li> <li>2. Значення <math>m</math> може змінюватися довільно від 1 до <math>n</math>.</li> <li>3. Сформувати початкову перестановку <math>(b[n], \dots, b[0])</math> таким чином, щоб кожен її елемент вибирався випадково.</li> </ol>
8	<p>Вивчити принципи роботи алгоритму генерації двійкових векторів довжини <math>n</math>. Написати програму генерації двійкових векторів довжини <math>n</math> у антилексикографічному порядку.</p> <ol style="list-style-type: none"> <li>1. Максимальне значення <math>n</math> дорівнює номеру залікової книжки (NZK).</li> <li>2. Значення <math>m</math> може змінюватися довільно від 1 до <math>n</math>.</li> <li>3. Сформувати початкову перестановку <math>(b[n], \dots, b[0])</math> таким чином, щоб кожен її елемент вибирався випадково.</li> </ol>
9	<p>Вивчити алгоритм генерації підмножин заданої множини, який використовує принципи роботи алгоритму генерації двійкових векторів. Написати програму генерації підмножин у лексикографічному порядку, використовуючи алгоритм генерації двійкових векторів.</p> <p>Вхідні параметри мають такі значення:</p>

№	Опис варіанта
	<p>1. Множину <math>\{a_0, a_1, \dots, a_{n-1}\}</math> сформувати з букв свого прізвища, імені та по батькові, виключивши повторення букв.</p> <p>2. Максимальне значення <math>n</math> дорівнює кількості одержаних різних букв з прізвища, імені та по батькові.</p> <p>3. Значення <math>m</math> може змінюватися довільно від 1 до <math>n</math>.</p>
10	<p>Вивчити алгоритм генерації підмножин заданої множини, який використовує принципи роботи алгоритму генерації двійкових векторів. Написати програму генерації підмножин у антилексикографічному порядку, використовуючи алгоритм генерації двійкових векторів. Вхідні параметри мають такі значення:</p> <p>1. Множину <math>\{a_0, a_1, \dots, a_{n-1}\}</math> сформувати з букв свого прізвища, імені та по батькові, виключивши повторення букв.</p> <p>2. Максимальне значення <math>n</math> дорівнює кількості одержаних різних букв з прізвища, імені та по батькові.</p> <p>3. Значення <math>m</math> може змінюватися довільно від 1 до <math>n</math>.</p>
11	<p>Вивчити алгоритм генерації підмножин заданої множини, який використовує принципи роботи алгоритму генерації двійкових векторів. Написати програму генерації підмножин імен у лексикографічному порядку, використовуючи алгоритм генерації двійкових векторів. Вхідні параметри мають такі значення:</p> <p>1. Множину <math>\{a_0, a_1, \dots, a_{n-1}\}</math> сформувати з імен своїх родичів та друзів загальною кількістю не менше 20.</p> <p>2. Максимальне значення <math>n</math> дорівнює потужності множини імен <math>n \geq 20</math>.</p> <p>3. Значення <math>m</math> може змінюватися довільно від 1 до <math>n</math>.</p>
12	<p>Вивчити алгоритм генерації підмножин заданої множини, який використовує принципи роботи алгоритму генерації двійкових векторів. Написати програму генерації підмножин імен у антилексикографічному порядку, використовуючи алгоритм генерації двійкових векторів. Вхідні параметри мають такі значення:</p> <p>1. Множину <math>\{a_0, a_1, \dots, a_{n-1}\}</math> сформувати з імен своїх родичів та друзів загальною кількістю не менше 20.</p> <p>2. Максимальне значення <math>n</math> дорівнює потужності множини імен <math>n \geq 20</math>.</p> <p>3. Значення <math>m</math> може змінюватися довільно від 1 до <math>n</math>.</p>
13	<p>Вивчити правила формування чисел у коді Грея та розглянути перший алгоритм генерації чисел у цьому коді. Написати програму генерації чисел у коді Грея за першим алгоритмом. Вхідні параметри мають такі значення:</p> <p>1. Початкове число <math>(b_1 b_2 \dots b_n)</math> сформувати шляхом переводу у двійкову систему числення числа, яке сформоване конкатенацією чисел, що відповідають вашому дню, місяцю та року народження.</p>

№	Опис варіанта
	<p><i>Приклад.</i> Дату 12 грудня 1998 перетворюємо у десяткове число 12121998, що відповідає двійковому числу 101110001111011110001110.</p> $12121998_{10} = 101110001111011110001110_2$ <p>2. Значення <math>n</math> дорівнює кількості розрядів одержаного двійкового числа.</p> <p><i>Приклад.</i> Для числа 101110001111011110001110 кількість розрядів <math>n = 24</math>.</p> <p>3. Значення <math>m</math> може змінюватися довільно від 1 до <math>n</math>.</p>
14	<p>Вивчити правила формування чисел у коді Грея та розглянути перший алгоритм генерації чисел у цьому коді. Написати програму генерації чисел у коді Грея за першим алгоритмом.</p> <p>1. Початкове число <math>(b_1 b_2 \dots b_n)</math> сформувати шляхом переведення у двійкову систему числення числа, яке сформоване конкатенацією чисел, що відповідають вашому року, місяцю та дню народження.</p> <p><i>Приклад.</i> Дату 1999 рік, місяць січень, 01 число перетворюємо у десяткове число 19990101, що відповідає двійковому числу 1001100010000011001010101.</p> $19990101_{10} = 1001100010000011001010101_2$ <p>2. Значення <math>n</math> дорівнює кількості розрядів одержаного двійкового числа.</p> <p><i>Приклад.</i> Для числа 1001100010000011001010101 кількість розрядів <math>n = 26</math>.</p> <p>3. Значення <math>m</math> може змінюватися довільно від 1 до <math>n</math>.</p>
15	<p>Вивчити спосіб формування підмножин з умовою мінімальної відмінності елементів. Використовуючи перший алгоритм генерації чисел у коді Грея написати програму генерації підмножин з умовою мінімальної відмінності елементів.</p> <p>Вхідні параметри мають такі значення:</p> <p>1. Базова множина для формування підмножин <math>\{a_1, a_2, \dots, a_n\}</math> складається з імен ваших родичів та друзів загальною кількістю не менше 20.</p> <p>2. <math>n</math> – потужність множини <math>\{a_1, a_2, \dots, a_n\}</math>.</p> <p>3. <math>m</math> – кількість множин, що потрібно згенерувати, може бути довільною від 1 до <math>n</math>.</p> <p>4. <math>(b_1 b_2 \dots b_n)</math> - початкове число у двійковій системі числення може бути довільним з кількістю розрядів <math>n</math>.</p>
16	<p>Вивчити спосіб формування підмножин з умовою мінімальної відмінності елементів. Використовуючи другий алгоритм генерації чисел у коді Грея написати програму генерації підмножин з умовою мінімальної відмінності елементів.</p> <p>Вхідні параметри мають такі значення:</p> <p>1. <math>A = (00, 01, 11, 10)</math> - початкова послідовність чисел, яка відповідає двохранрядним числам у коді Грея.</p>

№	Опис варіанта
	<p>2. <math>\{x_1, x_2, \dots, x_n\}</math> - базова множина для формування підмножин повинна складатися з не менше, ніж 20 назв міст України.</p> <p>3. <math>n</math> – потужність множини <math>\{x_1, x_2, \dots, x_n\}</math>.</p> <p>4. <math>(b_1 b_2 \dots b_n)</math> - початкове число у двійковій системі числення може бути довільним з кількістю розрядів <math>n</math>.</p>
17	<p>Вивчити принципи роботи алгоритму генерації сполучень з <math>n</math> по <math>k</math> в лексикографічному порядку. Написати програму генерації сполучень чисел десяткової системи числення у лексикографічному порядку.</p> <p>Вхідні параметри мають такі значення:</p> <p>1. Базова множина складається з послідовності чисел <math>(1, 2, 3, \dots, n)</math>, де <math>n \geq 32</math>.</p> <p>2. <math>(a_1, a_2, \dots, a_k)</math> - початкове сполучення, з якого алгоритм починає генерувати наступні сполучення у лексикографічному порядку, де <math>k</math> може приймати довільне значення від 1 до <math>n</math>.</p> <p>3. <math>r</math> – кількість сполучень, які необхідно згенерувати. Може приймати довільне значення від 1 до <math>C_n^k</math>.</p> <p><math>a_1 &lt; a_2 &lt; \dots &lt; a_k</math> - умова задавання початкового сполучення.</p>
18	<p>Вивчити принципи роботи алгоритму генерації сполучень з <math>n</math> по <math>k</math> в антилексикографічному порядку. Написати програму генерації сполучень чисел десяткової системи числення у антилексикографічному порядку.</p> <p>Вхідні параметри мають такі значення:</p> <p>1. Базова множина складається з послідовності чисел <math>(n, \dots, 3, 2, 1)</math>, де <math>n \geq 32</math>.</p> <p>2. <math>(a_1, a_2, \dots, a_k)</math> - початкове сполучення, з якого алгоритм починає генерувати наступні сполучення у лексикографічному порядку, де <math>k</math> може приймати довільне значення від 1 до <math>n</math>.</p> <p>3. <math>r</math> – кількість сполучень, які необхідно згенерувати. Може приймати довільне значення від 1 до <math>C_n^k</math>.</p> <p><math>a_1 &gt; a_2 &gt; \dots &gt; a_k</math> - умова задавання початкового сполучення.</p>
19	<p>Вивчити принципи роботи алгоритму генерації сполучень з <math>n</math> по <math>k</math> на множині. Написати програму генерації сполучень довільних елементів множини.</p> <p>Вхідні параметри мають такі значення:</p> <p>1. Базова множина <math>R</math> складається з проіндексованих елементів, якими є міста України.  <i>Приклад.</i> <math>r_1</math>=Київ, <math>r_2</math>=Харків, <math>r_3</math>=Дніпропетровськ і т. д.</p> <p>2. Загальна кількість елементів множини <math>n \geq 16</math>.</p>

№	Опис варіанта
	3. $k$ – кількість елементів сполучення може змінюватися довільно від 1 до $n$ .
20	<p>Вивчити принципи роботи алгоритму генерації сполучень з <math>n</math> по <math>k</math> на множині. Написати програму генерації у лексикографічному порядку сполучень довільних елементів множини.</p> <p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Базова множина <math>R</math> складається з проіндексованих елементів, якими є імена людей.</li> </ol> <p><i>Приклад.</i> <math>r_1</math>=Анастасія, <math>r_2</math>=Марія, <math>r_3</math>=Панас і т. д.</p> <ol style="list-style-type: none"> <li>2. Загальна кількість елементів множини <math>n \geq 16</math>.</li> <li>3. <math>k</math> – кількість елементів сполучення може змінюватися довільно від 1 до <math>n</math>.</li> <li>4. Передбачити можливість виводу обмеженої кількості підмножин.</li> </ol>
21	<p>Вивчити принципи роботи алгоритму генерації сполучень з <math>n</math> по <math>k</math> на множині. Написати програму генерації у антилексикографічному порядку сполучень довільних елементів множини.</p> <p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Базова множина <math>R</math> складається з проіндексованих елементів, якими є імена людей.</li> </ol> <p><i>Приклад.</i> <math>r_1</math>=Яків, <math>r_2</math>=Петро, <math>r_3</math>=Алла і т. д.</p> <ol style="list-style-type: none"> <li>2. Загальна кількість елементів множини <math>n \geq 16</math>.</li> <li>3. <math>k</math> – кількість елементів сполучення може змінюватися довільно від 1 до <math>n</math>.</li> <li>4. Передбачити можливість виводу обмеженої кількості підмножин.</li> </ol>
22	<p>Вивчити теоретичні основи та базові принципи роботи алгоритму генерації розбиття числа <math>n</math> у словниковому порядку. Написати програму для розбиття числа у словниковому порядку.</p> <p>Вхідні параметри мають такі значення:</p> <p>Число, яке підлягає розбиттю, може бути довільним числом від 1 до 100.</p>
23	<p>Вивчити принцип роботи алгоритму швидкого сортування QuickSort та алгоритм послідовного пошуку.</p> <p>Написати програму вибору з заданої множини людей певної підмножини імен людей, якщо відомий їх вік. Використати процедуру сортування QuickSort та функцію послідовного пошуку, яка починає пошук з максимального або мінімального віку з метою мінімізації кроків алгоритму.</p> <p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Базова множина складається з елементів, якими є імена людей та їх вік.</li> <li>2. Загальна кількість елементів множини <math>n \geq 16</math>.</li> <li>3. <math>k</math> – кількість показників віку, за якими потрібно провести вибір. <math>k</math> може змінюватися від 1 до <math>n</math>.</li> </ol>

№	Опис варіанта
	4. Передбачити вивід базової множини людей та їх вік, а також вибраної множини людей та їх вік.
24	<p>Вивчити принцип роботи алгоритму швидкого сортування Шелла та алгоритм послідовного пошуку.</p> <p>Написати програму вибору з заданої множини людей певної підмножини імен людей, якщо відомий їх вік. Використати процедуру сортування Шелла та функцію послідовного пошуку, яка починає пошук з максимального або мінімального віку з метою мінімізації кроків алгоритму.</p> <p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Базова множина складається з елементів, якими є імена людей та їх вік.</li> <li>2. Загальна кількість елементів множини <math>n \geq 16</math>.</li> <li>3. <math>k</math> – кількість показників віку, за якими потрібно провести вибір. <math>k</math> може змінюватися від 1 до <math>n</math>.</li> <li>4. Передбачити вивід базової множини людей та їх вік, а також вибраної множини людей та їх вік.</li> </ol>
25	<p>Вивчити принцип роботи алгоритму швидкого сортування QuickSort та алгоритм двійкового пошуку.</p> <p>Написати програму вибору з заданої множини людей певної підмножини імен людей, якщо відомий їх вік. Використати процедуру сортування QuickSort та функцію двійкового пошуку.</p> <p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Базова множина складається з елементів, якими є імена людей та їх вік.</li> <li>2. Загальна кількість елементів множини <math>n \geq 16</math>.</li> <li>3. <math>k</math> – кількість показників віку, за якими потрібно провести вибір. <math>k</math> може змінюватися від 1 до <math>n</math>.</li> <li>4. Передбачити вивід базової множини людей та їх вік, а також вибраної множини людей та їх вік.</li> </ol>
26	<p>Вивчити принцип роботи алгоритму швидкого сортування Шелла та алгоритм двійкового пошуку.</p> <p>Написати програму вибору з заданої множини людей певної підмножини імен людей, якщо відомий їх вік. Використати процедуру сортування Шелла та функцію двійкового пошуку.</p> <p>Вхідні параметри мають такі значення:</p> <ol style="list-style-type: none"> <li>1. Базова множина складається з елементів, якими є імена людей та їх вік.</li> <li>2. Загальна кількість елементів множини <math>n \geq 16</math>.</li> <li>3. <math>k</math> – кількість показників віку, за якими потрібно провести вибір. <math>k</math> може змінюватися від 1 до <math>n</math>.</li> <li>4. Передбачити вивід базової множини людей та їх вік, а також вибраної множини людей та їх вік.</li> </ol>



## ЗМІСТ

1.	<b>Лабораторна робота № 1.</b> «Множини: основні властивості та операції над ними, діаграми Венна».....	3
1.1.	Властивості множин	3
1.2.	Операції над множинами	5
1.3.	Діаграми Венна	7
1.4.	Тотожності алгебри множин	8
1.5.	Вимоги до програмного забезпечення	8
1.6.	Контрольні питання	9
1.7.	Блок-схеми алгоритмів виконання операцій над множинами	10
1.8.	Загальний порядок виконання лабораторної роботи	15
1.9.	Вимоги до інтерфейсу	15
1.8.	Індивідуальне завдання до лабораторної роботи №1	16
2.	<b>Лабораторна робота № 2.</b> «Бінарні відношення та їх основні властивості, операції над відношеннями» .....	19
2.1.	Основні означення	19
2.2.	Властивості бінарних відношень	20
2.3.	Операції над відношеннями	21
2.4.	Вимоги до програмного забезпечення	22
2.5.	Етапи виконання роботи	23
2.6.	Загальний порядок виконання лабораторної роботи	23
2.7.	Вимоги до інтерфейсу	24
2.8.	Варіанти для виконання лабораторної роботи	25
3.	<b>Лабораторна робота № 3.</b> «Графи. Способи представлення графів. Основні дерева. Пошук найкоротших шляхів».....	26
3.1.	Основні означення	26
3.2.	Матричні представлення	31
3.2.1.	Матриця суміжності	31
3.2.2.	Матриця інцидентності	32
3.3.	Найкоротший остов графа	33
3.4.	Алгоритм Прима-Краскала	33
3.5.	Контрольний приклад	34
3.6.	Задача про найкоротший шлях	35
3.6.1.	Алгоритм Дейкстри	36
3.6.2.	Алгоритм Форда-Беллмана знаходження мінімального шляху	43
3.6.3.	Алгоритм Флойда-Уоршела	48
3.6.4.	Метод динамічного програмування (топологічного сортування)	50
3.6.5.	Контрольний приклад	51
3.6.6.	Алгоритм пошуку шляхів у ширину	53
3.6.7.	Метод пошуку шляхів у глибину	56
3.6.8.	Алгоритм Левіта	58
3.7.	Вимоги до програмного забезпечення	60
3.8.	Варіанти для виконання лабораторної роботи	61

4.	<b>Лабораторна робота № 4 (додаткова).</b> «Розфарбовування графа, алгоритми розфарбовування» .....	63
4.1.	Основні означення	63
4.2.	Алгоритм прямого неявного перебору	65
4.3.	Приклад алгоритму прямого неявного перебору	66
4.4.	Евристичний алгоритм розфарбовування	67
4.5.	Приклад евристичного алгоритму розфарбовування	70
4.6.	Модифікований евристичний алгоритм розфарбовування	70
4.7.	Приклад модифікованого евристичного алгоритму розфарбовування	73
4.8.	Розфарбовування методом А. П. Єршова	74
4.9.	Приклад розфарбовування методом А. П. Єршова	76
4.10.	Рекурсивна процедура послідовного розфарбовування	80
4.11.	Приклад роботи рекурсивної процедури послідовного розфарбовування	81
4.12.	«Жадібний» алгоритм розфарбовування	82
4.13.	Приклад роботи «жадібного» алгоритму розфарбовування	84
4.14.	Завдання до лабораторної роботи №4	85
4.15.	Вимоги до програмного забезпечення	86
4.16.	Варіанти для виконання лабораторної роботи	87
5.	<b>Лабораторна робота № 5 (додаткова).</b> «Комбінаторика: перестановки, розміщення, сполучення».....	89
5.1.	Основні означення	89
5.2.	Комбінаторні алгоритми	91
5.2.1.	Алгоритм побудови перестановок у лексикографічному порядку	92
5.2.2.	Блок-схема алгоритму побудови перестановок у лексикографічному порядку	96
5.2.3.	Алгоритм генерації двійкових векторів довжини $n$	98
5.2.4.	Блок-схема алгоритму генерації двійкових векторів довжини $n$	98
5.2.5.	Алгоритм генерації підмножин заданої множини	100
5.2.6.	Блок-схема алгоритму генерації підмножин заданої множини	101
5.2.7.	Перший алгоритм генерації коду Грея	102
5.2.8.	Блок-схема першого алгоритму генерації коду Грея	103
5.2.9.	Другий алгоритм генерації коду Грея	105
5.2.10.	Блок-схема другого алгоритму генерації коду Грея	105
5.2.11.	Перший алгоритм генерації підмножин з умовою мінімальної відмінності елементів	106
5.2.12.	Блок-схема першого алгоритму генерації підмножин з умовою мінімальної відмінності елементів.	107
5.2.13.	Другий алгоритм генерації підмножин з умовою мінімальної відмінності елементів	108
5.2.14.	Блок-схема другого алгоритму генерації підмножин з умовою мінімальної відмінності елементів.	109
5.2.15.	Алгоритм генерації сполучень з $n$ по $k$ в лексикографічному порядку.	110

5.2.16.	Блок-схема алгоритму генерації сполучень з $n$ по $k$ в лексикографічному порядку	112
5.2.17.	Алгоритм генерації сполучень з $n$ по $k$ на множині.	114
5.2.18.	Блок-схема алгоритму генерації сполучень з $n$ по $k$ на множині.	114
5.2.19.	Алгоритм генерації розбиття числа $n$ у словниковому порядку	116
5.2.20.	Блок-схема алгоритму генерації розбиття числа $n$ у словниковому порядку	119
5.2.21.	Швидке сортування QuickSort	120
5.2.22.	Сортування Шелла	121
5.2.23.	Методи пошуку	123
5.3.	Завдання лабораторної роботи	125
5.4.	Вимоги до програмного забезпечення	126
5.5.	Варіанти для виконання лабораторної роботи	127