МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ім. ІГОРЯ СІКОРСЬКОГО"

А.В. Яковенко

Алгоритмізація та програмування

*Затверджено Вченою радою НТУУ "КПІ ім. Ігоря Сікорського"
як підручник для студентів які навчаються за спеціальностями
163 "Біомедична інженерія" та
152 "Метрологія та інформаційно-вимірювальна техніка"
спеціалізаціями "Клінічна інженерія, Реабілітаційна інженерія,
Біомедична інформатика" та
"Біомедичні прилади та інформаційні системи"*

Київ
КПІ ім. Ігоря Сікорського
2018

ЗМІСТ

Комп'ютерний практикум №1 Програмування в графічному режимі **..... 3**

Комп'ютерний практикум №2 Рекурсія **..... 6**

Комп'ютерний практикум №3 Метод Карацуби множення двох цілих чисел **. 9**

Комп'ютерний практикум №4 Файли **..... 10**

Комп'ютерний практикум №5 Алгоритми лінійного та бінарного пошуку **.... 14**

Комп'ютерний практикум №6 Алгоритм сортування злиттям **..... 16**

Комп'ютерний практикум №7 Підрахунок інверсій **..... 20**

Комп'ютерний практикум №8 Швидке сортування **..... 23**

Комп'ютерний практикум № 9 Алгоритми на графах **..... 25**

Комп'ютерний практикум № 10 Алгоритм Дейкстри **..... 33**

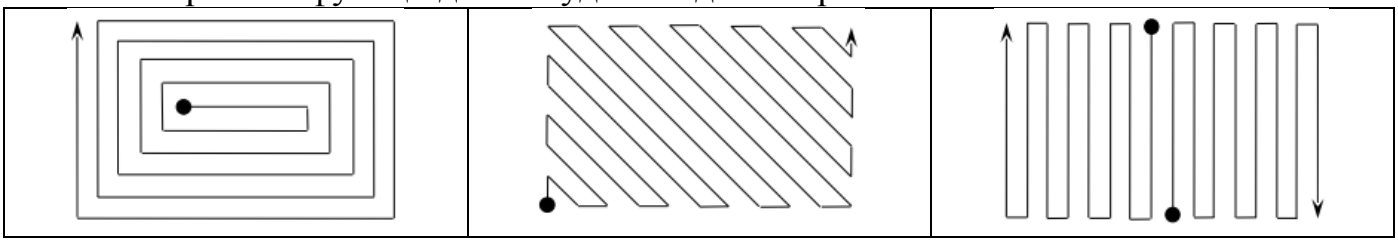
Комп'ютерний практикум №1

Програмування в графічному режимі

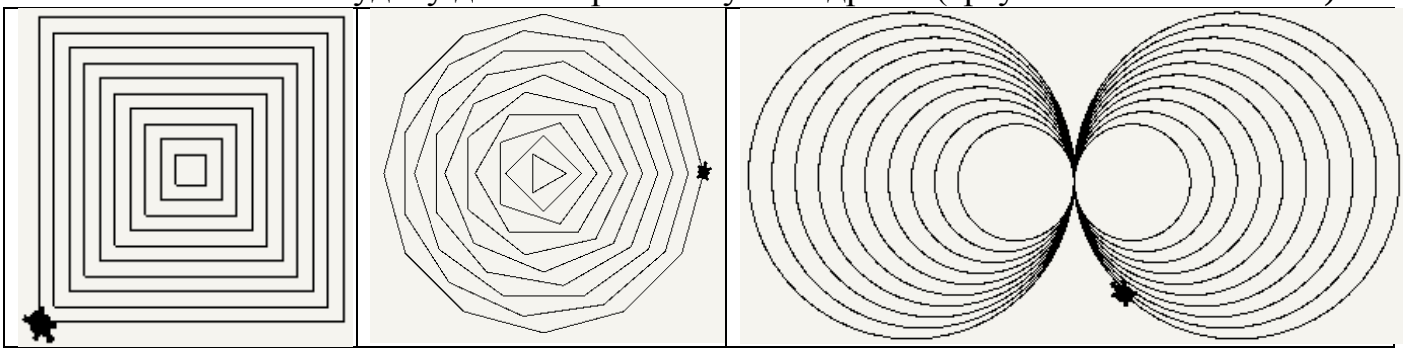
Мета роботи: написання програм для формування графічних зображень з використанням вбудованого модулю Python.

Завдання

1. Розробити функції для побудови заданих орнаментів.



2. Виконати побудову даного орнаменту з квадратів (аргументи – кількість ***n***).



Примітка. Правильний багатокутник – це багатокутник з рівними сторонами і кутами. Кут між двома сусідніми вершинами правильного n-кутника дорівнює:

$$\alpha = \frac{360°}{n}$$

Теоретичні відомості

ПРОГРАМУВАННЯ В ГРАФІЧНОМУ РЕЖИМІ (МОДУЛЬ TURTLE)

В Python розроблено кілька модулів, що забезпечують роботу з графікою. Два графічних модуля є частиною стандартної бібліотеки Python:

- ***turtle*** (черепашка) – простий графічний пакет, який так само може бути використаний для створення нескладного графічного інтерфейсу – Graphical User Interface (GUI);

- ***tkinter*** – розроблений безпосередньо для створення графічного інтерфейсу користувача GUI. Так, інтерфейс IDLE побудований з використанням tkinter.

Комп'ютерна графіка – це досить велика і складна область знань, що включає знання про технічні засоби, що дозволяють відображати зображення. Знання про способи формування кольорового зображення, яке сприймається або як світіння окремих точок дисплея, або як відображення, наприклад, від аркуша паперу: адитивна кольорова модель RGB або субтрактивна – CMY. Велика область математичних і фізичних знань допомагає вирішувати питання переміщення, повороту об'єкта, формування другого і першого планів зображення (сцени), обліку рефлексів (вторинних відображень) і ще багато чого.

При побудові графіків використовуються функції та методи графічної системи, які дозволяють:

- формувати вікно, в якому буде будуватися графік (розмір, система координат);

- будувати графічні примітиви (точка, лінія, ...);

- задавати ширину і колір ліній, колір фону, виконувати заливку зображення або його частини заданим кольором;

- управляти маркером (покажчиком), який використовується для малювання;

- наносити текст.

МОДУЛЬ TURTLE

Модуль ***turtle*** входить в стандартну поставку бібліотеки Python.

Модель цього модуля наступна. Є прямокутна поверхня, по якій повзає черепашка. Черепашка може переміщатися на задану відстань прямо, назад, під кутом або за заданими координатами. Черепашку можна клонувати, створюючи групу черепашок. При цьому кожна черепашка живе своїм життям. Для малювання черепашка використовує колірне перо (олівець), яке може бути піднято або опущено. Якщо перо опущено, то залишається слід. Можна змінювати колір і товщину лінії (Табл. 1).

Черепашка розуміє команди, за допомогою яких можна намалювати коло заданого радіусу і кольору, дугу з заданим кутом, залити фігуру певним кольором, отримати поточний стан налаштувань або змінити їх. Форма черепашки може бути змінена користувачем і використана як штамп, після якого на полотні залишається малюнок.

Таблиця 1. Функції та методи модуля turtle

Метод	**Значення**
forward(x)	Пройти вперед x пікселів
backward(x)	Пройти назад x пікселів
left(x)	Повернутися наліво на x градусів
right(x)	Повернутися направо на x градусів
penup()	Не залишати слід при русі
pendown()	Залишати слід при русі
shape(x)	Змінити значок черепахи ("arrow", "turtle", "circle", "square", "triangle", "classic")
stamp()	Намалювати копію черепахи в поточному місці
color()	Встановити колір
begin_fill()	Необхідно викликати перед малюванням фігури, яку треба зафарбувати
end_fill()	Викликати після закінчення малювання фігури
width()	Встановити товщину лінії
goto(x, y)	Перемістити черепашку в точку (x, y)
setx(x)	Встановити x координату черепашки
sety(y)	Встановити y координату черепашки
setheading(x)	Повернути черепашку під кутом x до вертикалі (0 – вгору, 90 – направо)
home()	Повернути черепашку додому - в точку, з координатами (0,0)
circle(radius)	Намалювати коло радіуса | r |, центр якої знаходиться зліва від черепашки, якщо r> 0 і справа, якщо r <1
speed(speed)	Встановити швидкість черепашки. speed має бути від 1 (повільно) до 10 (швидко), або 0 (миттєво)

Наприклад, щоб намалювати літеру S:

```
import turtle

def letter(length):
    turtle.shape('turtle')
    turtle.forward(length)
    turtle.left(90)
    turtle.forward(length)
    turtle.left(90)
    turtle.forward(length)
    turtle.right(90)
    turtle.forward(length)
    turtle.right(90)
    turtle.forward(length)
```

Наприклад, щоб намалювати спіраль, черепашка повинна повертатися на незмінну величину і рухатися вперед на постійно збільшувану відстань:

Необхідний ***while***, умова якого завжди буде виконуватися (тобто нескінченний цикл), з припиненням виконання після досягнення черепашкою певної відстані від центру. Використовучи функцію ***turtle.distance (x, y)***, отримуємо відстань від черепашки до заданої точки.

Ще знадобляться функції ***turtle.xcor ()*** і ***turtle.ycor ()***, які повертають координати (черепашки) ***X*** і ***Y*** відповідно.

```
import turtle

def draw_spiral(radius):
    original_xcor = turtle.xcor()
    original_ycor = turtle.ycor()
    speed = 1
    turtle.shape("turtle")
    while True:
        turtle.forward(speed)
        turtle.left(10)
        speed += 0.1
        if turtle.distance(original_xcor, original_ycor) > radius:
            break
```

Комп'ютерний практикум №2

Рекурсія

Мета роботи: реалізація рекурсивних алгоритмів.

Завдання

1. Використовуючи рекурсивну функцію, розрахувати значення суми:

$$S = \frac{1+1}{2+5} + \frac{2+1}{4+5} + \frac{3+1}{6+5} + \cdots \frac{n+1}{2n+5}$$

2. Задані значення $\boldsymbol{x}$, точність $\boldsymbol{\varepsilon}$. Скласти програму розрахунку функції $\boldsymbol{y}$ з точністю $\boldsymbol{\varepsilon}$, використовуючи рекурсивний та ітераційний алгоритми розв'язання задачі. Визначити, яку кількість членів ряду необхідно підсумувати для досягнення зазначеної точності (порівняти результат підсумовування зі значенням стандартної функції).

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}$$

Теоретичні відомості

РЕКУРСІЯ

В мові програмування Python функція може викликати будь-яку кількість інших функцій. Функції також можуть викликати самі себе, тобто мають властивість рекурсивності.

Рекурсія – спосіб опису об'єктів або обчислювальних процесів через самих себе. Рекурсивне програмування дозволяє описати процес що повторюється без явного використання операторів циклу.

Багато математичних функцій можна описати рекурсивно. Класичним прикладом програмування рекурсії є задача знаходження $n!$.

Відомо, що $n!$ можна представити у вигляді:

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$$

Тобто

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

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

Інший приклад рекурсивної функції для піднесення числа до цілої додатньої степені x^n:

$$x^n = \begin{cases} 1 & n = 0 \\ x * x^{n-1} & n > 0 \end{cases}$$

```
def rec_func (n):
    if  n>0:
        return x* rec_func (n-1)
    else:
        return 1
```

Рекурсивна функція обов'язково повинна містити хоча б одну альтернативу, що не використовує рекурсивний виклик, тобто явне визначення для деяких значень аргументів функції, тобто умову виходу (закінчення рекурсивності), щоб не спричинити зациклення програми. Кожний (новий) виклик вимагає додаткової пам'яті з ресурсу програмного стека. Якщо кількість викликів (глибина рекурсії) надмірно велика, виникає переповнення сегмента стека і операційна система вже не може створити наступний примірник локальних об'єктів функції, що як правило, веде до аварійного завершення програми.

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}$$

```
def rec_func_2(n, i):
    if  i==n:
        return 1/n
    else:
        return 1/i+rec_func_2(n, i+1)
```

Можна простежити, як працює функція ***rec_func_2***, наприклад, для ***n = 5***.
rec_func_2(5,1);
При виконанні тіла функції сформується наступне:

$$\mathbf{\frac{1}{1}} + \boldsymbol{rec\_func\_2}(\mathbf{5}, \mathbf{2})$$

Що знову змушує звернутися до функції ***rec_func_2(5,2)***, що призводить до появи нового значення:

$$\mathbf{\frac{1}{2}} + \boldsymbol{rec\_func\_2}(\mathbf{5}, \mathbf{3})$$

Після виконання ще двох звернень ситуація виявиться наступною:

$$\mathbf{\frac{1}{3}} + \boldsymbol{rec\_func\_2}(\mathbf{5}, \mathbf{4})$$

$$\mathbf{\frac{1}{4}} + \boldsymbol{rec\_func\_2}(\mathbf{5}, \mathbf{5})$$

Потім при черговому виклику функції ***rec_func_2(5,5)*** рекурсивні звернення припиняться і буде повернено значення $\frac{1}{5}$. В результаті сформується така послідовність:

Значення $\mathbf{\frac{1}{5}}$ буде передано до $\mathbf{\frac{1}{4}} + \boldsymbol{rec\_func\_2}(\mathbf{5}, \mathbf{5})$ замість $\boldsymbol{rec\_func\_2}(\mathbf{5}, \mathbf{5})$, потім $\mathbf{\frac{1}{4}} + \mathbf{\frac{1}{5}}$ до $\mathbf{\frac{1}{3}} + \mathbf{rec\_func\_2}(\mathbf{5}, \mathbf{4})$ замість $\boldsymbol{rec\_func\_2}(\mathbf{5}, \mathbf{4})$ і т.д.

В результаті отримаємо ряд:

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}$$

Ця послідовність операторів і дає результат обчислення суми:
Сума =2.28333333

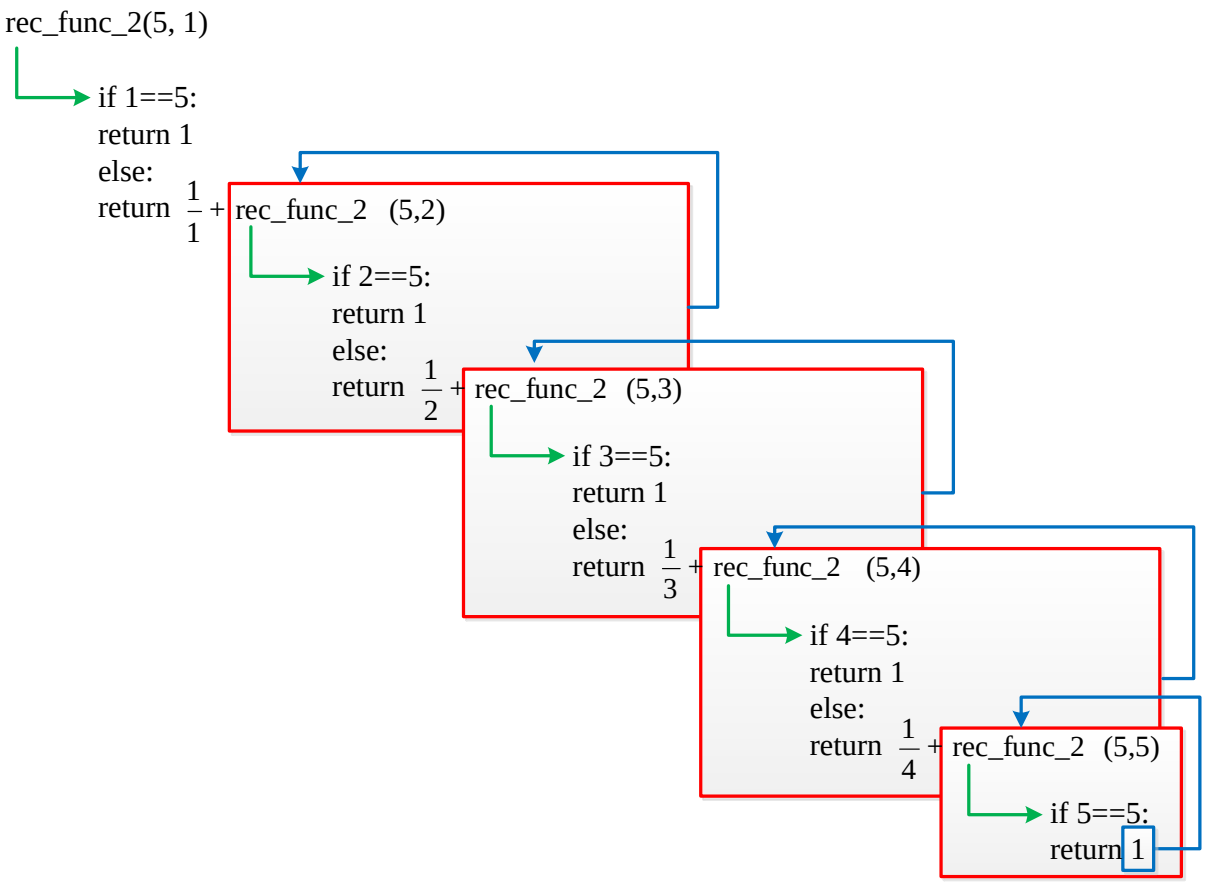


Рис.2.1. Графічне зображення роботи рекурсії

Ітерація і рекурсія засновані на керуючих структурах: ітерація використовує структуру повторення, рекурсія використовує структуру розгалуження.

```
def non_rec_func(n):
    S=0
    for i in range(1, n+1):
        S+=1/i
    return S

print(non_rec_func(5))
```

І ітерація, і рекурсія передбачають повторення: ітерація використовує структуру повторення явно, рекурсія – за допомогою повторних викликів функції.

Ітерація і рекурсія включають перевірку на завершення: ітерація завершується, коли перестає виконуватися умова продовження циклу, рекурсія завершується, коли розпізнається нерекурсивний випадок.

Ітерація з її перевіркою повторення продовжує виконувати тіло циклу, поки умова продовження циклу не буде порушено. Рекурсія продовжує виробляти більш прості варіанти початкової задачі, поки не буде досягнутий нерекурсивний випадок.

І ітерація, і рекурсія може відбуватися нескінченно: ітерація потрапляє в нескінченний цикл, якщо умова продовження циклу ніколи не стає хибною; рекурсія триває нескінченно, якщо крок рекурсії не редукує задачу таким чином, що задача сходиться до нерекурсивного випадку.

Будь-яка проблема, яка може бути вирішена рекурсивно, може бути також вирішена і ітераційно (не рекурсивно).

Комп'ютерний практикум №3

Метод Карацуби множення двох цілих чисел

Мета роботи: реалізація алгоритму множення чисел з довгої арифметики.

Завдання

1. На вхід подається два числа X та Y великої розрядності. Реалізовувати метод Карацуби мовою Python для заданих двох чисел.

Для спрощення реалізації можна припустити, що розрядності всіх вхідних чисел в тестах дорівнюють ступеням двійки, тобто: 2, 4, 8, 16, 32, 64, 128, ...

2. Підрахувати кількість появ певних значень цієї суми **ad+bc** протягом всієї роботи методу над заданими двома числами.

Теоретичні відомості

МЕТОД КАРАЦУБИ МНОЖЕННЯ ДВОХ ЦІЛИХ ЧИСЕЛ

Нехай, в нас є два числа X та Y, які мають розрядність n, кожне з яких ми можемо розбити навпіл і отримати пари чисел a,b та c,d відповідно, кожне з яких буде мати розрядність $n/2$. В такому випадку можна записати: $X=10^{n/2}a+b$ та $Y=10^{n/2}c+d$.

Тоді добуток $X\cdot Y=10^{n}ac+10^{n/2}(ad+bc)+bd$.

Основна специфіка методу полягає в тому, щоб обраховувати не чотири менші добутки ac, ad, bc, bd, а три – ac, bd, $(a+b)(c+d)$, де добуток $(a+b)(c+d)$ використовується для обрахунку $ad+bc$ із використанням вже обрахованих сум ac, bd: $ad+bc=(a+b)(c+d)-ac-bd$.

Окрім того, що сам метод Карацуби працює швидше за стандартний метод множення в стовпчик, його можна успішно застосовувати для множення чисел у завданнях з, так званою, довгою арифметикою, коли розрядність чисел може сягати 100 і більше.

Наприклад, при розрядності чисел 50:

X = 21625695688898558125310188636840316594920403182768

Y = 13306827740879180856696800391510469038934180115260

XY = 28776940730884664097031015150982625548257536241915584289131190955687867000042535211298788108583968 0

ВИРІВНЮВАННЯ ЧИСЕЛ

Хоча вхідні числа можуть мати розмірність кратну степеням 2, але під час роботи методу в рекурсивних викликах можуть траплятись випадки, коли два множники мають різну розмірність або вона однакова, але не кратна двом.

Наприклад, нехай маємо числа X=12345,Y=6789. Можна розглянути три варіанти для визначення чому буде дорівнювати a, b, c, d:

1. Зробити два числа однакової довжини і вирівняти їх до довжини, яка є степенем 2.

2. Зробити два числа однакової довжини і вирівняти їх до довжини, яка є кратним 2.

3. Не робити вирівнювання взагалі, але тоді треба бути уважним зі зведенням коефіцієнтів ac,bd,ad+bc в один результат

Комп'ютерний практикум №4

Файли

Мета роботи: організація роботи з файлами.

Завдання

1. Розробити програму предбачивши збереження даних, що вводяться в файл і можливість читання з раніше збереженого файлу. Результати виводити на екран і в текстовий файл.

Файл ***input.txt*** містить кілька рядків тексту. Слова в тексті можуть розділятися пробілами та розділовими знаками. Переписати текст в файл ***output.txt***, видаливши слова, в яких кількість приголосних букв менше кількості голосних.

Теоретичні відомості

ФАЙЛИ

Активна програма працює з даними, які зберігаються в запам'ятовуючому пристрої з довільним доступом (Random Access Memory (RAM)). RAM – дуже швидка пам'ять, але вона дорога і вимагає постійного живлення; якщо живлення пропаде, то всі дані, які в ній зберігаються, будуть втрачені. Жорсткі диски повільніші оперативної пам'яті, але вони більш місткі, коштують дешевше і можуть зберігати дані навіть після того, як хтось вимкне живлення. Тому багато зусиль при створенні комп'ютерних систем направлено на пошук кращого співвідношення між зберіганням даних на диску і в оперативній пам'яті.

Найпростіший приклад стійкого сховища – це файл, що являє собою послідовність байтів, яка зберігається під ім'ям файлу. Можна зчитувати дані з файлу в пам'ять і записувати дані з пам'яті в файл. Python дозволяє робити це досить легко.

Перед тим як щось записати в файл або зчитати з нього, необхідно відкрити його. Щоб відкрити файл, програма повинна викликати функцію ***open()***, передавши їй ім'я зовнішнього файлу і режим роботи:

fileobj = open (filename, mode)

де

- ***fileobj*** – це об'єкт файлу, що повертається функцією ***open ()***;
- ***filename*** - це рядок, що представляє собою ім'я файлу;
- ***mode*** - це рядок, що вказує на тип файлу і дії, які необхідно над ним зробити.

Перша літера рядка ***mode*** вказує на операцію:

- ***r*** означає читання;
- ***w*** означає запис. Якщо файлу не існує, він буде створений. Якщо файл існує, він буде перезаписаний;
- ***x*** означає запис, але тільки якщо файлу ще не існує;
- ***a*** означає додавання даних в кінець файлу, якщо він існує.

Друга літера рядка ***mode*** вказує на тип файлу:

- ***t*** (або нічого) означає, що файл текстовий;
- ***b*** означає, що файл бінарний.

Крім того, функція може приймати третій необов'язковий аргумент, керуючий буферизацією виведених даних, – значення нуль означає, що вихідна інформація не буде буферизованою (тобто вона буде записуватися у зовнішній файл відразу ж, в момент виклику методу запису). Ім'я зовнішнього файлу може включати

платформозалежні префікси абсолютного або відносного шляху до файлу. Якщо шлях до файлу не вказано, передбачається, що він знаходиться в поточному робочому каталозі (тобто в каталозі, де був запущений сценарій).

Після відкриття файлу можна викликати функції для читання або запису даних. По завершенню роботи, файл потрібно закрити.

Існує кілька різновидів методів читання і запису, а в табл. 2 перераховані лише найбільш часто використовувані з них.

Таблиця 2 Методи для роботи з файлами

Операція	**Інтерпретація**
output = open(r'C:\spam', 'w')	Відкриває файл для запису ('w' означає write – запис)
input = open('data', 'r')	Відкриває файл для читання ('R' означає read – читання)
input = open('data')	Те ж саме, що і в попередньому рядку (Режим 'r' використовується за умовчанням)
aString = input.read()	Читання файлу цілком в єдиний рядок
aString = input.read(N)	Читання наступних N символів (або байтів) в рядок
aString = input.readline()	Читання наступного текстового рядка (включаючи символ кінця рядка) в рядок
aList = input.readlines()	Читання файлу цілком в список рядків (включаючи символ кінця рядка)
output.write(aString)	Запис рядка символів (або байтів) в файл
output.writelines(aList)	Запис всіх рядків зі списку в файл
output.close()	Закриття файлу вручну (виконується по закінченні роботи з файлом)
output.flush()	Виштовхує вихідні буфери на диск, файл залишається відкритим
anyFile.seek(N)	Змінює поточну позицію в файлі для наступної операції, зміщуючи її на N байтів від початку файлу.
for line in open('data'): операции над line	Ітерації по файлу, порядкове читання
open('f.txt', encoding='latin-1')	Файли з текстом Юникода в Python 3.0 (Рядки типу str)
open('f.bin', 'rb')	Файли з двійковими даними в Python 3.0 (Рядки типу bytes)

Якщо необхідно записати дані до файлу, його потрібно відкрити в режимі для запису, записати інформацію, після чого файл закривається. Якщо текстового файлу не існувало, він буде створений та записаний. У випадку, якщо файл існував, то файл буде перезаписаний.

```
myfile = open('myfile.txt', 'w')
myfile.write('hello text file\n')
myfile.write('goodbye text file\n')
```

```
myfile.close()
```

Для порядкового зчитування, файл відкривається в режимі для читання, використовується метод ***readline().***

```
myfile = open('myfile.txt')
print(myfile.readline())
print(myfile.readline())
myfile.close()
```

Якщо необхідно вивести вміст файлу, його слід прочитати в рядок цілком, за допомогою методу ***read()***, і вивести:

```
myfile = open('myfile.txt')
print(myfile.read())
print(myfile.read())

myfile.close()
```

Якщо необхідно переглянути вміст файлу рядок за рядком, можна використати ітератор файлу:

```
myfile = open('myfile.txt')
for line in myfile:
   print(line, end='')

myfile.close()
```

В цьому випадку створюється об’єкт файлу, вміст якого автоматично буде читатися ітератором і повертатися по одному рядку в кожній ітерації циклу.

В мові Python існує підтримка текстових та двійкових (бінарних) файлів:

- Вміст текстових файлів представляється у вигляді звичайних рядків типу ***str***, виконується автоматичне кодування/декодування символів Юнікоду і за замовчуванням проводиться інтерпретація символів кінця рядка.

- Вміст двійкових файлів представляється у вигляді рядків типу ***bytes***, і передається програмі без будь-яких змін.

Крім того, так як текстові файли реалізують автоматичне перетворення символів Юнікоду, не можна відкрити файл з двійковими даними в текстовому режимі – перетворення його вмісту в символи Юнікоду, швидше за все, завершиться помилкою.

Дані завжди записуються в файл у вигляді рядків, а методи запису не виконують автоматичного форматування рядків:

```
X, Y, Z = 43, 44, 45
myfile = open('myfile.txt', 'w')
myfile.write('%s,%s,%s\n' % (X, Y, Z))

myfile.close()
```

Щоб виконати зворотні перетворення та отримати з рядків у текстовому файлі дійсні об’єкти мови Python, необхідно виконати відповідні перетворення, щоб

можна було використовувати операції над цими об’єктами, такі як індексування, складання і так далі. Інтерпретатор Python ніколи автоматично не виконує перетворення рядків у числа або в об’єкти інших типів.

```
myfile = open('myfile.txt')
line = myfile.readline()
line.rstrip()                          # видалення символу кінця рядку
parts = line.split(',')                # розбиття на підрядки по комам

myfile.close()
```

Було застосовано метод ***rstrip()***, щоб видалити завершальний символ кінця рядка.

В результаті було отримано список рядків, кожен з яких містить окреме число. Тепер щоб перетворити ці рядки в цілі числа для подальших математичних операцій над ними:

```
numbers = [int(P) for P in parts]
```

Щоб кожного разу вручну не виконувати закриття файлу, можна скористатися ***менеджером контексту*** (в основному застосовуються для обробки винятків, проте вони дозволяють обробляти програмний код, що виконує операції з файлами, додатковим шаром логіки, який гарантує, що після виходу за межі блоку інструкцій менеджера файл буде закритий автоматично, і дозволяє не покладатися на автоматичне закриття файлів механізмом збірки сміття):

```
with open(r'C:\mis\ myfile.txt') as myfile:
    for line in myfile:
        ...операції над рядком line...
```

Оригінальний текст помітно спростився, тому що звільнення ресурсів в цьому випадку відбувається автоматично (всередині менеджера контексту).

У момент виклику функції ***open()*** Python шукає вказаний файл в поточному робочому каталозі. В момент запуску програми поточний робочий каталог там, де збережена програма.

Визначити поточний робочий каталог можна наступним чином:

```
>>> import os
>>> os.getcwd()
'D:\\Users\\Desktop'
```

Якщо файл знаходиться в іншому каталозі, то необхідно вказати шлях до нього:
1. абсолютний шлях (починаючи з кореневого каталогу):

```
'C:\\Users\\Student\\data1.txt'
```

2. відносний шлях (щодо поточного робочого каталогу):

```
'data\\data1.txt'
```

Комп'ютерний практикум №5

Алгоритми лінійного та бінарного пошуку

Мета роботи: реалізація алгоритмів лінійного та бінарного пошуку в масивах.

Завдання

1. Реалізувати алгоритм лінійного пошуку мовою Python для заданих вхідних даних.

2. Реалізувати алгоритм бінарного пошуку мовою Python для заданих вхідних даних.

Вхідні дані

У першому рядку вхідних даних містяться натуральні числа ***N*** і ***K*** (***0<N, K≤100000***). У другому рядку задаються ***N*** елементів першого масиву, відсортованого по зростанню, а в третьому рядку – ***K*** елементів другого масиву. Елементи обох масивів – цілі числа, кожне з яких по модулю не перевищує 10^9.

Вихідні дані

Потрібно для кожного з ***K*** чисел вивести в окремий рядок "YES", якщо це число зустрічається в першому масиві, і "NO" у протилежному випадку.

Приклад

Вхідні дані:
10 5
1 2 3 4 5 6 7 8 9 10
-2 0 4 9 12
Вихідні дані:
NO
NO
YES
YES
NO

Теоретичні відомості

АЛГОРИТМ ЛІНІЙНОГО ПОШУКУ

Лінійний пошук виконує пошук елемента або значення з деякої множини до тих пір, поки потрібний елемент або значення не буде знайдений, порядок пошуку послідовний.

Порівнюється шуканий елемент з усіма елементами на деякій множині, якщо шуканий елемент буде знайдений, то функція повертає індекс знайденого елементу, в іншому випадку функція повертає ***-1***. Лінійний пошук застосовується до невідсортованих або неупорядкованих множин.

Алгоритм у вигляді псевдокоду що виконує лінійний пошук в масиві:

```
for i←1 to n do
   if a[i]=k then
      return i
   else
      return -1
```

АЛГОРИТМ БІНАРНОГО ПОШУКУ

Двійковий (бінарний) пошук елемента в масиві застосується лише до відсортованих масивів. Розглянемо, як здійснюється бінарний пошук у відсортованому за зростанням масиві. Під впорядкованим масивом будемо розуміти масив, впорядкований за неспаданням, тобто $a_1 \le a_2 \le \ldots \le a_n$.

Нехай, маємо задану своїми межами область пошуку. Обираємо її середину та, якщо шуканий елемент менший, ніж середній, то пошук здійснюємо у лівій частині, інакше – в правій. Дійсно, якщо шуканий елемент менший середнього, то і менший всіх елементів, які знаходяться правіше середнього, а значить, їх відразу можна виключити із розгляду. Аналогічно для випадку, коли шуканий елемент більший середнього.

Алгоритм у вигляді псевдокоду що виконує бінарний пошук в упорядкованому масиві:

```
while left < right – 1 do
   mid← (left+right)/2
   if a[mid] > key then
      right ← mid
   else
      left ← mid
if left ≥ 0 and a[left] = x then
   return left
else
   return -1
```

Перед виконанням необхідно змінним ***left*** та ***right*** привласнити значення - ∞ та + ∞, відповідно. У випадку якщо елемент не знайдений алгоритм повертає – 1.

Складність алгоритму бінарного пошуку складає O(logn), де ***n*** – кількість елементів масиву.

Комп'ютерний практикум №6

Алгоритм сортування злиттям

Мета роботи: отримання практичних навичок в обробці масивів, у сортуванні методом злиття.

Завдання

На вхід подається масив з n елементів (input_1000.txt). Елементи розділені пробілом, перше число – кількість елементів (з сортування виключити).

Необхідно реалізувати алгоритм сортування злиттям та визначити час роботи реалізованої функції.

Теоретичні відомості

АЛГОРИТМ СОРТУВАННЯ ЗЛИТТЯМ

Основною операцією методу сортування злиттям є об'єднання двух відсортованих послідовностей під час комбінування. Це робиться за допомогою виклику допоміжної процедури ***Merge(A, p, q, r)***, де ***A*** – масив, а ***p, q, r*** – індекси, що нумерують елементи масиву, такі що ***p≤q<r***. В цій процедурі припускається, що елементи підмасивів ***A[p..q]*** та ***A[q+1..r]*** впорядковані. Процедура зливає цих два впорядкованих підмасиви в один відсортований, елементи якого замінюють поточні елементи підмасивів ***A[p.. r]***.

Процедура працює наступним чином. Уявімо перед нами стоїть дві шеренги солдат, де в кожній солдати розміщені за зростом у порядку зростання. Цих дві шеренги необхідно об'єднати в одну, в якій солдати будуть стояти правильно за зростанням. Необхідно подивитися на першого солдата з першої шеренги та першого солдата з другої шеренги і порівняти між собою їх. Той солдат, що має менший зріст буде першим у вихідній шерензі. При цьому для наступного порівняння ми віьмемо знову тепер вже першого солдата з однієї з шеренг і першого солдата з іншої. Цей крок повторюється до тих пір доки одна з шерен не зпорожніє. Після чого ті солдати що залишилися просто переходять в вихідну шеренгу.

Описана ідея реалізована у псевдокоді. Проте для її спрощення використовуються додаткові міркування. Щоб на кожному кроці не перевіряти, чи не спорожнів якийсь з двох підмасивів, до кожного з них додається так званий сигнальний елемент, який має значення нескінченності ∞. Таким чином, не існує елементів масивів, які були б більшими за ці сигнальні елементи. Робота процедури Merge продовжується до тих пір, поки поточні елементи в обох підмасивах не виявляться сигнальними. Як тільки це станеться, це буде означати, що всі несигнальні елементи розміщені у вихідний масив. Оскільки завчасно відомо, що у вихідному масиві повинен бути присутній ***r – p + 1*** елемент, то виконавши таку кількість кроків можна зупинитись.

```
Merge(A, p, q, r)
n1 ← q – p + 1
n2 ← r – q
Створити масиви L[1..n1+1] та R[1..n2+1]
for i ← 1 to n1 do
   L[i] ← A[p+i-1]
for j ← 1 to n2 do
   R[j] ← A[q+j]
```

```
L[n1+1] ← ∞
R[n2+1] ← ∞
i ← 1
j ← 1
for k ← p to r do
   if L[i] ≤ R[j] then
      A[k] ← L[i]
      i ← i + 1
   else
      A[k] ← R[j]
      j ← j + 1
```

Детально опишемо роботу процедури Merge. В рядку 1 обраховується довжина ***n1*** підмасиву ***A[p…q]***, а у рядку 2 - довжина ***n2*** підмасиву ***A[q+1…r]***. Далі в рядку 3 створюються масиви ***L*** («лівий») та ***R*** («правий»), довжини яких відповідно ***n1 + 1*** та ***n2 + 1***. В двох циклах for в рядках 4-5 та 6-7 елементи масиву ***A[p…q]*** та ***A[q+1…r]*** копіюються у масиви ***L*** та ***R*** відповідно. В рядках 8 та 9 останнім елементам масивів ***L*** та ***R*** приписуються сигнальні значення.

Як показано на рис. 3.1, в результаті копіювання та додавання сигнальних елементів отримуємо масив ***L*** з послідовністю чисел ***⟨2, 4, 5, 7, ∞⟩*** та масив ***R*** з послідовністю ***⟨1, 2, 3, 6, ∞⟩***. Світло-сірі комірки масиву ***A*** містять кінцеві елементи, а світло-сірі комірки масивів ***L*** та ***R*** – значення, які ще тільки необхідно скопіювати в масив ***A***. У темно-сірих комірках ***A*** містяться значення, які будуть замінені іншими, а в темносірих масивів ***L*** та ***R*** – значення, які вже скопійовані назад в ***A***.

В рядках 10-17 лістингу 3.1 виконуються ***r – p + 1*** основних кроків, в ході кожного з яких відбуваються маніпуляції з інваріантом циклу:

Перед кожною ітерацією циклу for в рядках 12-17, підмасив ***A[p…k–1]*** містить ***k–p*** найменших елементів масивів ***L*** та ***R*** у відсортованому порядку. Окрім того, елементи ***L[i]*** та ***R[j]*** є найменшими елементами ***L*** та ***R***, які ще не були скопійовані у ***A***.

Необхідно показати, що цей інваріант циклу зберігається перед першою ітерацією даного циклу for, що кожна ітерація циклу не порушує його, і що з його допомогою можна продемонструвати коректність алгоритму, коли цикл закінчує свою роботу.

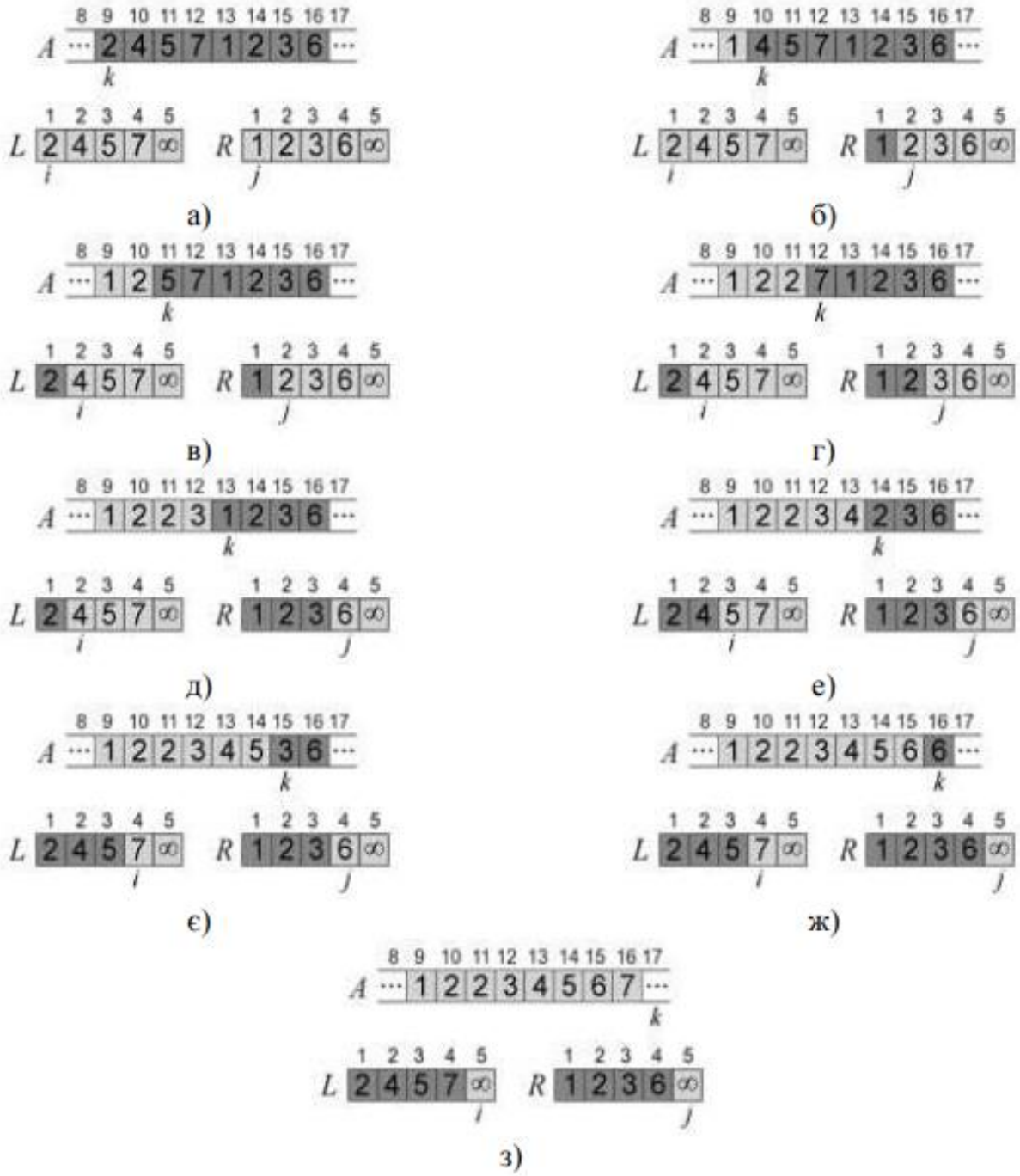


Рис. 6.1. Приклад виконання процедури Merge

Тепер процедуру Merge можна використовувати в якості підпрограми в алгоритмі сортування злиттям. Процедура ***MergeSort(A, p, r)*** виконує сортування елементів в підмасиві ***A[p…r]***. Якщо виконується нерівність ***p≥r***, то в цьому масиві елементів міститься не більше одного, тому він є відсортованим. У протилежному випадку відбувається розбиття, під час якого обраховується індекс q, який розбиває масив ***A[p…r]*** на два підмасиви: ***A[p…q]*** з $\lceil n/2 \rceil$ елементами та ***A[q+1…r]*** з $\lfloor n/2 \rfloor$ елементами.

```
MergeSort(A, p, r)
if p<r then
   q ← ⌊(p + r)/2⌋
```

```
MergeSort(A, p, q)
MergeSort(A, q+1, r)
Merge(A, p, q, r)
```

Щоб відсортувати послідовність $\boldsymbol{A} = \langle \boldsymbol{A}[\mathbf{1}], \boldsymbol{A}[\mathbf{2}], \dots, \boldsymbol{A}[\boldsymbol{n}] \rangle$, викликається процедура ***MergeSort(A, 1, length[A])***, де ***length[A]*** = $\boldsymbol{n}$. На рис. 3.2 наводиться приклад роботи цієї процедури, якщо n – ступінь двійки. В ході роботи відбувається попарне об'єднання одноелементних послідовностей у відсортовані послідовності довжини 2, потім – попарне об'єднання двоелементних послідовностей у відсортовані послідовності довжини 4 і т.д., допоки не будуть отримані дві послідовності довжиною n/2, які об'єднаються у кінцеву відсортовану послідовність довжиною n.

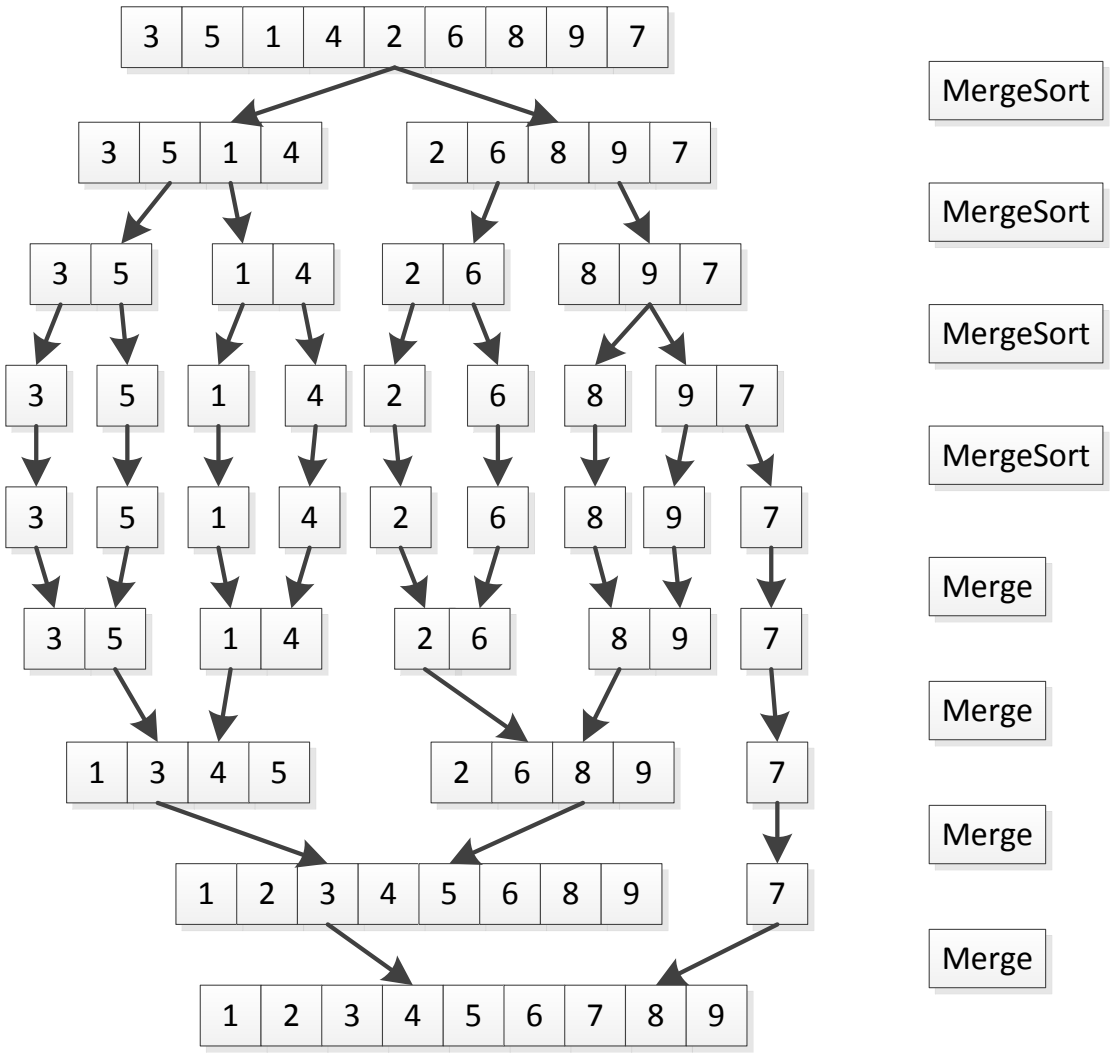


Рис 6.2. Процес сортування масиву A =$\langle 3, 5, 1, 4, 2, 6, 8, 9, 7 \rangle$

Комп'ютерний практикум №7

Підрахунок інверсій

Мета роботи: метод декомпозиції. Сортування злиттям

Завдання

1. На вхід подається матриця з n користувачами та m фільмами (input_1000_5.txt).

Необхідно визначити кількість відносних інверсій для двох вказаних користувачів. Номери користувачів індексуються з 1. В роботі використати реалізацію алгоритму сортування злиттям з попередньої лабораторної.

Теоретичні відомості

ПІДРАХУНОК ІНВЕРСІЙ

Припустимо, що ***A[1...n]*** – це масив, який складається з n різних чисел. Якщо i < j та ***A[i] > A[j]***, то пара ***(i, j)*** називається інверсією в масиві ***A***. Задача полягає в тому, щоб знайти кількість всіх інверсій в заданому масиві ***A***. Наприклад, нехай заданий масив ***A = ⟨2, 3, 8, 6, 1⟩***. Тоді пара індексів (1, 5) буде інверсією, адже ***A[1] = 2*** та ***A[5] = 1*** і ***2>1***. Загалом масив ***A*** містить наступні інверсії: ***(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)*** і їх кількість – ***5***. Якщо масив відсортований (наприклад, ***A = ⟨1, 2, 3, 6, 8⟩***), то він не містить жодної інверсії. У випадку, коли масив відсортований у зворотному порядку, то кількість інверсій максимальна і дорівнює $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ (так для масиву ***A = ⟨8, 6, 3, 2, 1⟩*** кількість інверсій становитиме ***10***). Одним з прикладів застосування інверсій є обрахунок того, наскільки два пріоритетні списки є подібними один до одного.

В рамках підходу «розділяй та володарюй», за аналогією з алгоритмом методу злиття, масив можна розбивати на дві частини. Тоді всі інверсії ***(i, j)***, де ***i <j***, будуть підпадати під одну з трьох категорій:

1) ліві інверсії: якщо ***i, j ≤ n/2***,

2) праві інверсії: якщо ***i, j > n/2***,

3) розділені інверсії: якщо ***i ≤ n/2 < j***.

Так, у масиві ***A = ⟨2, 3, 8, 6, 1⟩*** буде жодної лівої інверсії – підмасив ***AL = ⟨2, 3, 8⟩*** немає інверсій, буде одна права інверсія – підмасив A_R ***= ⟨6, 1⟩***, та 4 розділені інверсії.

Тоді на етапі рекурсивного розв'язку методу «розділяй та володарюй» необхідно обрахувати кількість лівих та правих інверсій. Розділені інверсії повинні обраховуватись на етапі комбінування.

```
CountInv(A)
   n ← length(A)
   if n=1 then
      return 0
   else
      x ← CountInv(перша половина A)
      y ← CountInv(друга половина A)
      z ← CountSplitInv(A)
      return x+y+z
```

Нехай перший та другий підмасиви A відсортовані всередині. Масив ***A = ⟨1, 3, 5, 2, 4, 6⟩***, в якому ліва частина ***L = ⟨1, 3, 5⟩*** та права частина ***A = ⟨2, 4, 6⟩*** вже відсортовані (рис. 1, а). Випадки, коли вставляється елемент лівого масиву в масив ***A***, відповідають тій ситуації, при якій поточний елемент для вставки ***L[i]*** є меншим за ***R[j]*** та у початковому масиві ***A*** знаходився лівіше за будь-який елемент підмасиву ***R*** (випадки б, г та е рис. 1). В таких випадках жодних розділених інверсій з елементом ***L[i]*** не може існувати.

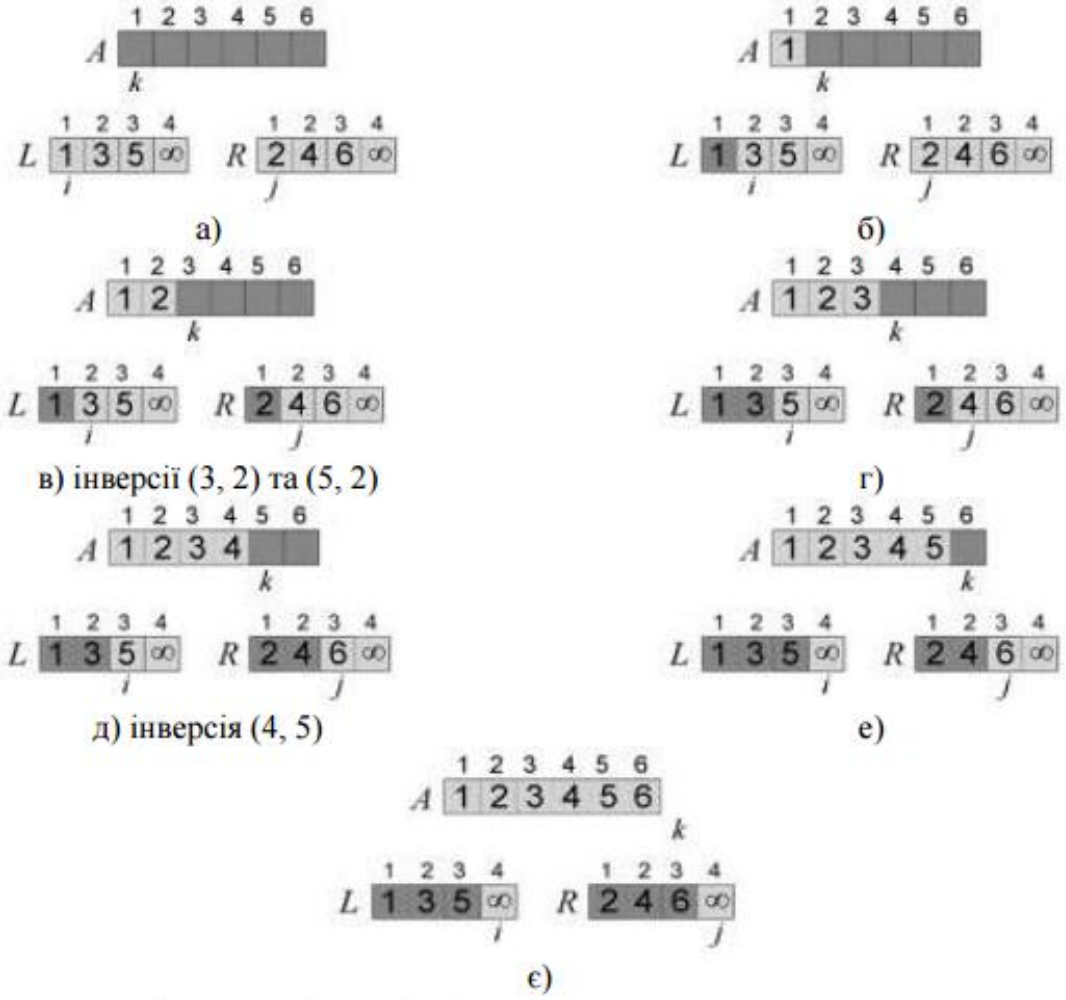


Рис. 7.1. Підрахунок інверсій під час роботи процедури Merge для масиву A = ⟨1, 3, 5, 2, 4, 6⟩

Якщо вставляється елемент правого підмасиву ***R*** у масив ***A***, то це означає, що найменший не вставлений елемент ***R[j]*** є також меншим за деякі елементи, що знаходяться у лівому підмасиві ***L***, та ще не були вставлені у масив ***A***. Всі ці невставлені елементи з ***L*** будуть більшими ***R[j]*** і, отже, будуть створювати з ним інверсії. Кількість таких інверсій буде дорівнювати кількості ще не доданих до ***A*** елементів з підмасиву ***L***.

Після модифікації процедуру ***CountInv***, яка використовує сортування злиттям всередині себе, позначимо як ***SortAndCountInv***.

```
SortAndCountInv(A)
   n ← length(A)
   if n=1 then
      return (A, 0)
```

```
else
   (L, x) ← SortAndCountInv(перша половина A)
    (R, y) ← SortAndCountInv(друга половина A)
    (A, z) ← MergeAndCountSplitInv(A, L, R)
    return (A, x+y+z)
```

Процедура ***SortAndCountInv*** приймає на вхід масив ***A***, в якому необхідно обрахувати кількість інверсій. На виході процедури буде відсортований масив ***A*** та знайдена кількість інверсій в ньому. Рядок 3 процедури ***SortAndCountInv*** відповідає базовому випадку, коли масив ***A*** містить тільки один елемент. В цьому випадку масив вже є відсортованим та не містить інверсій. Якщо масив містить більше ніж один елемент, то тоді виконуються рядки 4-7. В рядку 4 рекурсивно викликається процедура ***SortAndCountInv*** для лівої половини масиву ***A***. Результатом цього виклику буде цей відсортований підмасив, а також кількість лівих інверсій у ***A***. Аналогічно результатом виконання рядку 6 буде відсортована права частина масиву ***A*** разом із кількістю правих інверсій в ***A***. Тепер лишається об'єднати лівий ***L*** та правий ***R*** підмасиви ***A*** і підрахувати кількість розділених інверсій в ***A***. Це робиться у процедурі ***MergeAndCountSplitInv***, якій передаються відсортовані підмасиви ***L*** та ***R***.

```
MergeAndCountSplitInv(A, L, R)
   n1 ← length(L)
   n2 ← length(R)
   L[n1+1] ← ∞
   R[n2+1] ← ∞
   i ← 1
   j ← 1
   c ← 0                                   // лічильник розділених інверсій
   for k ← p to r do
      if L[i] ≤ R[j] then
         A[k] ← L[i]
         i ← i + 1
      else
         A[k] ← R[j]
          j ← j + 1
         c ← c + (n2 – i + 1)
   return (A, c)
```

Змінна ***c*** на виході буде містити кількість розділених інверсій в масиві ***A***. Лічильник ***c*** збільшується на кількість неопрацьованих елементів в масиві ***L*** (передостанній рядок процедури).

Комп'ютерний практикум №8

Швидке сортування

Мета роботи: реалізація алгоритму швидкого сортування.

Завдання

1. Запрограмувати алгоритм швидкого сортування таким чином:

- щоб опорним елементом в процедурі ***Partition*** обирався останній елемент поточного масиву;

- щоб опорним елементом в процедурі ***Partition*** обирався перший елемент поточного масиву (достатньо поміняти місцями перший та останній елементи та викликати процедуру ***Partition***, яка працює з останнім елементом в якості опорного);

- щоб опорним елементом в процедурі ***Partition*** обирається медіана серед трьох елементів поточного масиву: першого, останнього та середнього (елементи за індексами ***p, r*** та $\lfloor \frac{p+r}{2} \rfloor$).

Медіаною є середній за значенням елемент серед множини елементів. Наприклад, якщо масив містить елементи ***8 2 4 5 7 1***, то необхідно розглянути перший (8), останній (1) та середній (4) елемент; медіаною в цьому випадку буде елемент 4, який має індекс 3 у вхідному масиві.

2. Підрахувати загальну кількість порівнянь елементів в процедурі розбиття під час роботи алгоритму над масивом, який заданий у вхідному файлі (input__10000.txt).

Не потрібно враховувати у загальне порівняння елементів ті порівняння, які відбуваються при визначенні медіани.

Коли досягнуто випадка, коли поточний масив містить лише два елементи. Тоді так само викликаємо процедуру розділення з медіаною, просто в якості середнього автоматично буде обраний перший елемент масиву.

Теоретичні відомості

ШВИДКЕ СОРТУВАННЯ

Швидке сортування (quick sort) – це алгоритм сортування, час роботи якого для вхідного масиву з n чисел в найгіршому випадку дорівнює $\Theta(n^2)$. Не дивлячись на таку повільну роботу в найгіршому випадку, цей алгоритм на практиці часто виявляється оптимальним завдяки тому, що в середньому час його роботи набагато кращий: $\Theta(nlgn)$.

Процес сортування підмасиву ***A[p…r]*** складається з трьох етапів:

– Розділення: Масив ***A[p…r]*** розбивається на два (можливо порожніх) підмасиви ***A[p…q–1]*** та ***A[q+1…r]*** шляхом переупорядкування його елементів. Кожний елемент масиву ***A[p…q–1]*** не є більшим за елемент ***A[q]***, а кожний елемент підмасиву ***A[q+1…r]*** є більшим елементу ***A[q]***. Індекс ***q*** обраховується під час процедури розбиття.

– Рекурсивний розв'язок: Підмасиви ***A[p…q–1]*** та ***A[q+1…r]*** відсортовуються шляхом рекурсивного виклику процедури швидкого сортування.

– Комбінування: Оскільки підмасиви відсортовуються на місці без застосування додаткової пам'яті, для їх об'єднання не потрібні жодні додаткові дії: увесь масив ***A[p…r]*** виявляється відсортованим.

Процедура сортування ***QuickSort.***

```
QuickSort(A, p, r)
   if p<r then
      q ← Partition(A, p, r)
      QuickSort(A, p, q-1)
      QuickSort(A, q+1, r)
```

Щоб виконати сортування всього масиву ***A***, виклик процедури повинен мати вигляд ***QuickSort(A, 1, length[A])***.

Ключовою частиною алгоритму швидкого сортування є процедура ***Partition***, яка змінює порядок елементів підмасиву ***A[p…r]*** без використання додаткової пам'яті.

```
Partition(A, p, r)
   x ← A[r]
   i ← p - 1
   for j ← p to r-1 do
      if A[j] ≤ x then
         i ← i + 1
         Обміняти A[i] ↔ A[j]
   Обміняти A[i+1] ↔ A[r]
   return i + 1
```

По суті, процедура ***Partition*** виконує те, що описано вище в пункті «Розбиття» парадигми «розділяй та володарюй». На початку процедури обирається опорний (pivot) елемент. Таким елементом буде останній елемент масиву ***A[p…r]*** – елемент ***x = A[r]***. Після цього всі елементи масиву ***A[p…r – 1]*** розбиваються на такі, що не більше опорного – вони будуть розташовані згодом ліворуч від нього, та такі, які не менше нього і будуть розташовані праворуч. В кінці елемент ***A[r]*** переміщується в позицію, яка відповідає цьому розділенню: всі елементи ліворуч не більше нього та всі елементи праворуч – не менше.

В процесі роботи масив ***A[p…r]*** складається з чотирьох частин (деякі з них можуть бути порожніми) (рис. 1), які утворюють інваріант циклу:

1) якщо $p \le k \le i$, то $A[k] \le x$;
2) якщо $i + 1 \le k \le j - 1$, то $A[k] > x$;
3) якщо $k = r$, то $A[k] = x$;
4) якщо $j \le k \le r - 1$, то елементи можуть мати довільні значення.

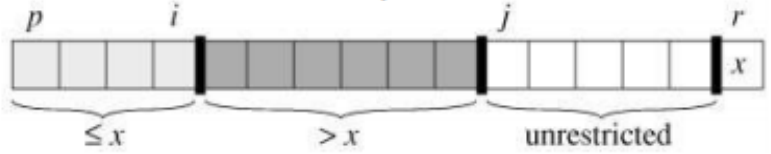


Рис. 8.1. Області масиву A[p…r], які підтримуються процедурою Partition.

Комп'ютерний практикум № 9

Алгоритми на графах

Мета роботи: ознайомитися з алгоритмами на графах.

Завдання

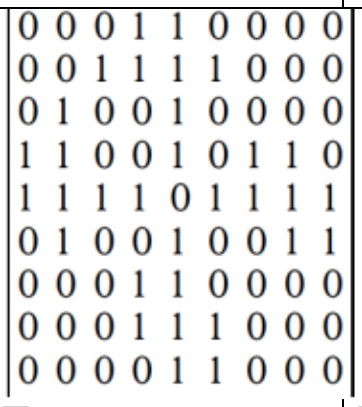
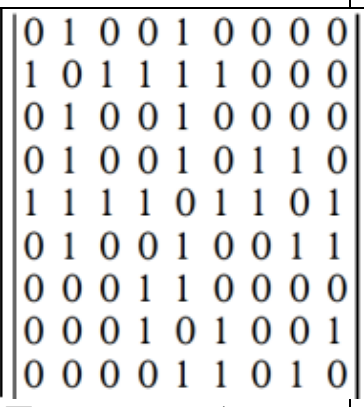
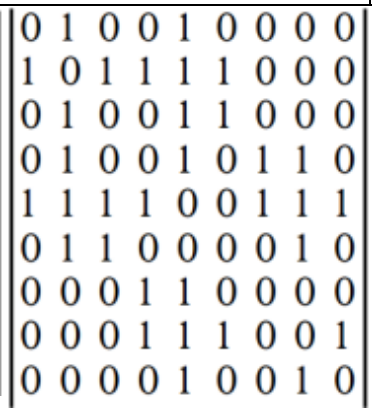
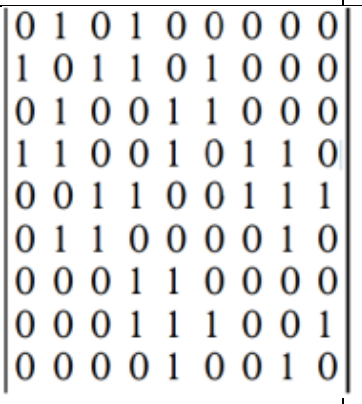
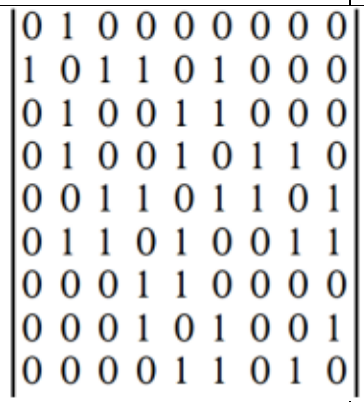
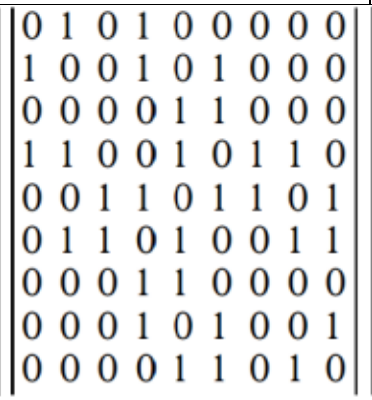
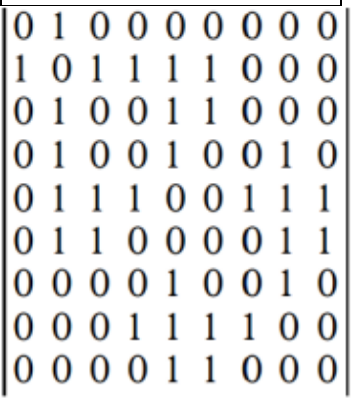
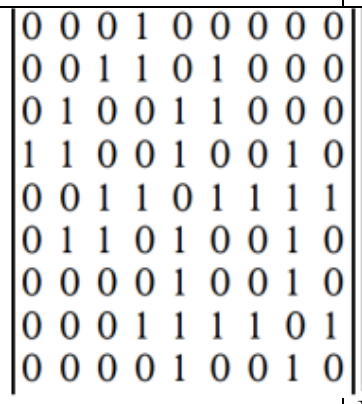
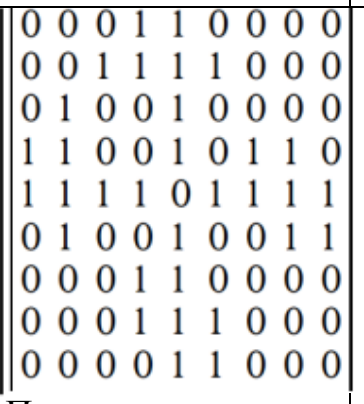
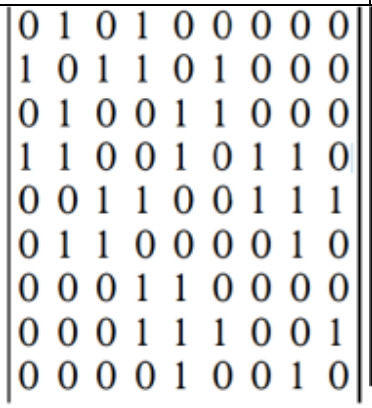
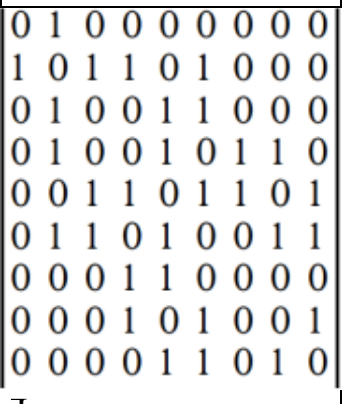
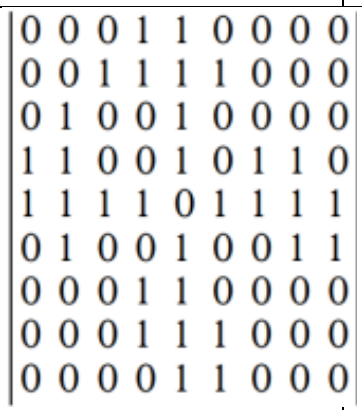
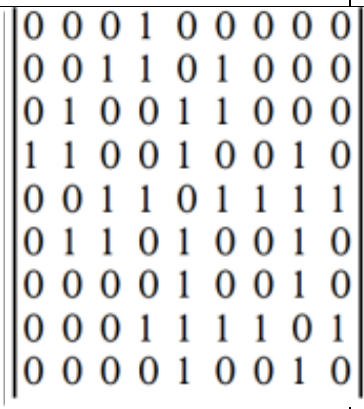
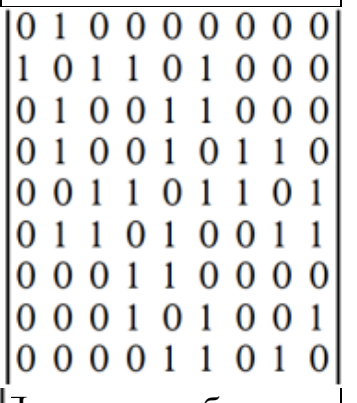
За заданою матрицею суміжності визначити таку послідовність з'єднаних вершин графа з вершини n_1 в n_2, в якій буде мінімальна кількість ребер.

Вершини n_1 і n_2 вибрати самостійно.

Застосувати до свого варіанту процедуру пошуку в глибину або процедуру пошуку в ширину і розглянути чи існує шлях з вершини ***A*** у вершину ***B***, які вводить користувач.

Варіанти завдань:

1	2	3	4
$\begin{vmatrix} 0&1&0&1&0&0&0&0&0 \\ 1&0&0&1&0&1&0&0&0 \\ 0&0&0&0&1&1&0&0&0 \\ 1&1&0&0&1&0&1&1&0 \\ 0&0&1&1&0&1&1&0&1 \\ 0&1&1&0&1&0&0&1&1 \\ 0&0&0&1&1&0&0&0&0 \\ 0&0&0&1&0&1&0&0&1 \\ 0&0&0&0&1&1&0&1&0 \end{vmatrix}$	$\begin{vmatrix} 0&1&0&0&0&0&0&0&0 \\ 1&0&1&1&1&1&0&0&0 \\ 0&1&0&0&1&1&0&0&0 \\ 0&1&0&0&1&0&0&1&0 \\ 0&1&1&1&0&0&1&1&1 \\ 0&1&1&0&0&0&0&1&1 \\ 0&0&0&0&1&0&0&1&0 \\ 0&0&0&1&1&1&1&0&0 \\ 0&0&0&0&1&1&0&0&0 \end{vmatrix}$	$\begin{vmatrix} 0&0&0&1&0&0&0&0&0 \\ 0&0&1&1&0&1&0&0&0 \\ 0&1&0&0&1&1&0&0&0 \\ 1&1&0&0&1&0&0&1&0 \\ 0&0&1&1&0&1&1&1&1 \\ 0&1&1&0&1&0&0&1&0 \\ 0&0&0&0&1&0&0&1&0 \\ 0&0&0&1&1&1&1&0&1 \\ 0&0&0&0&1&0&0&1&0 \end{vmatrix}$	$\begin{vmatrix} 0&0&0&1&1&0&0&0&0 \\ 0&0&1&1&1&1&0&0&0 \\ 0&1&0&0&1&0&0&0&0 \\ 1&1&0&0&1&0&1&1&0 \\ 1&1&1&1&0&1&1&1&1 \\ 0&1&0&0&1&0&0&1&1 \\ 0&0&0&1&1&0&0&0&0 \\ 0&0&0&1&1&1&0&0&0 \\ 0&0&0&0&1&1&0&0&0 \end{vmatrix}$
Пошук в глибину	Пошук в ширину	Пошук в глибину	Пошук в ширину
5	6	7	8
$\begin{vmatrix} 0&1&0&0&1&0&0&0&0 \\ 1&0&1&1&1&1&0&0&0 \\ 0&1&0&0&1&0&0&0&0 \\ 0&1&0&0&1&0&1&1&0 \\ 1&1&1&1&0&1&1&0&1 \\ 0&1&0&0&1&0&0&1&1 \\ 0&0&0&1&1&0&0&0&0 \\ 0&0&0&1&0&1&0&0&1 \\ 0&0&0&0&1&1&0&1&0 \end{vmatrix}$	$\begin{vmatrix} 0&1&0&0&1&0&0&0&0 \\ 1&0&1&1&1&1&0&0&0 \\ 0&1&0&0&1&1&0&0&0 \\ 0&1&0&0&1&0&1&1&0 \\ 1&1&1&1&0&0&1&1&1 \\ 0&1&1&0&0&0&0&1&0 \\ 0&0&0&1&1&0&0&0&0 \\ 0&0&0&1&1&1&0&0&1 \\ 0&0&0&0&1&0&0&1&0 \end{vmatrix}$	$\begin{vmatrix} 0&1&0&1&0&0&0&0&0 \\ 1&0&0&1&0&1&0&0&0 \\ 0&0&0&0&1&1&0&0&0 \\ 1&1&0&0&1&0&1&1&0 \\ 0&0&1&1&0&1&1&0&1 \\ 0&1&1&0&1&0&0&1&1 \\ 0&0&0&1&1&0&0&0&0 \\ 0&0&0&1&0&1&0&0&1 \\ 0&0&0&0&1&1&0&1&0 \end{vmatrix}$	$\begin{vmatrix} 0&1&0&1&0&0&0&0&0 \\ 1&0&1&1&0&1&0&0&0 \\ 0&1&0&0&1&1&0&0&0 \\ 1&1&0&0&1&0&1&1&0 \\ 0&0&1&1&0&0&1&1&1 \\ 0&1&1&0&0&0&0&1&0 \\ 0&0&0&1&1&0&0&0&0 \\ 0&0&0&1&1&1&0&0&1 \\ 0&0&0&0&1&0&0&1&0 \end{vmatrix}$
Пошук в глибину	Пошук в ширину	Пошук в глибину	Пошук в ширину
9	10	11	12
$\begin{vmatrix} 0&1&0&0&0&0&0&0&0 \\ 1&0&1&1&0&1&0&0&0 \\ 0&1&0&0&1&1&0&0&0 \\ 0&1&0&0&1&0&1&1&0 \\ 0&0&1&1&0&1&1&0&1 \\ 0&1&1&0&1&0&0&1&1 \\ 0&0&0&1&1&0&0&0&0 \\ 0&0&0&1&0&1&0&0&1 \\ 0&0&0&0&1&1&0&1&0 \end{vmatrix}$	$\begin{vmatrix} 0&1&0&1&0&0&0&0&0 \\ 1&0&0&1&0&1&0&0&0 \\ 0&0&0&0&1&1&0&0&0 \\ 1&1&0&0&1&0&1&1&0 \\ 0&0&1&1&0&1&1&0&1 \\ 0&1&1&0&1&0&0&1&1 \\ 0&0&0&1&1&0&0&0&0 \\ 0&0&0&1&0&1&0&0&1 \\ 0&0&0&0&1&1&0&1&0 \end{vmatrix}$	$\begin{vmatrix} 0&1&0&0&0&0&0&0&0 \\ 1&0&1&1&1&1&0&0&0 \\ 0&1&0&0&1&1&0&0&0 \\ 0&1&0&0&1&0&0&1&0 \\ 0&1&1&1&0&0&1&1&1 \\ 0&1&1&0&0&0&0&1&1 \\ 0&0&0&0&1&0&0&1&0 \\ 0&0&0&1&1&1&1&0&0 \\ 0&0&0&0&1&1&0&0&0 \end{vmatrix}$	$\begin{vmatrix} 0&0&0&1&0&0&0&0&0 \\ 0&0&1&1&0&1&0&0&0 \\ 0&1&0&0&1&1&0&0&0 \\ 1&1&0&0&1&0&0&1&0 \\ 0&0&1&1&0&1&1&1&1 \\ 0&1&1&0&1&0&0&1&0 \\ 0&0&0&0&1&0&0&1&0 \\ 0&0&0&1&1&1&1&0&1 \\ 0&0&0&0&1&0&0&1&0 \end{vmatrix}$
Пошук в ширину	Пошук в глибину	Пошук в ширину	Пошук в глибину

13	14	15	16
			
Пошук в ширину	Пошук в глибину	Пошук в ширину	Пошук в ширину
17	18	19	20
			
Пошук в глибину	Пошук в ширину	Пошук в глибину	Пошук в ширину
21	22	23	24
			
Пошук в глибину	Пошук в ширину	Пошук в глибину	Пошук в ширину
25	26	27	28
			
Пошук в глибину	Пошук в ширину	Пошук в ширину	Пошук в глибину

Теоретичні відомості

АЛГОРИТМИ НА ГРАФАХ

Математичні моделі у вигляді графів широко використовуються при моделюванні різноманітних явищ, процесів і систем. Як результат, багато теоретичних і реальних прикладних завдань можуть бути вирішені за допомогою тих чи інших процедур аналізу графових моделей. Серед множини цих процедур може бути виділений деякий певний набір типових алгоритмів обробки графів.

Є два стандартних способи представлення графа ***G = (V, E)***: як набір списків суміжних вершин або як матриці суміжності. Обидва способи представлення застосовні як для орієнтованих, так і для неорієнтованих графів.

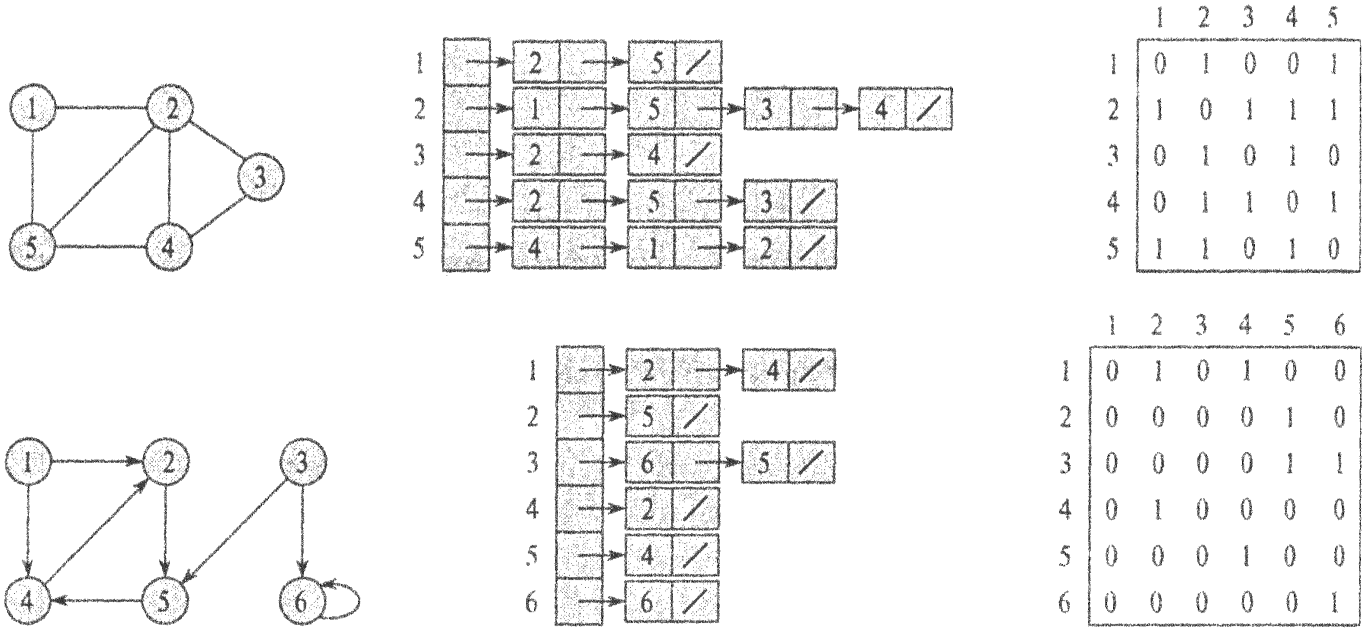


Рис. 9.1. Два представлення орієнтованого графа

Зазвичай більш переважним є представлення за допомогою списків суміжності, оскільки воно забезпечує компактне уявлення розріджених (sparse) графів, тобто таких, для яких $|E|$ набагато менше $|V|^2$. Подання за допомогою матриці суміжності краще в разі сильно зв'язних (dense) графів, тобто коли значення $|E|$ близьке до $|V|^2$, або коли треба мати можливість швидко визначити, чи є ребро, що з'єднує дві дані вершини.

Подання графа ***G = (V, E)*** у вигляді списку суміжності (adjacency-list representation) використовує масив ***Adj*** з $|V|$ списків, по одному для кожної вершини з ***V***. Для кожної вершини $u \in V$ список ***Adj [u]*** містить всі вершини v, такі що $(u, v) \in E$, тобто ***Adj[u]*** складається з усіх вершин, суміжних з ***u*** в графі ***G*** (список може містити і не самі вершини, а покажчики на них). Вершини в кожному списку зазвичай зберігаються в довільному порядку.

ПОШУК ВШИР

Пошук в ширину (breadth-first search) являє собою один з найпростіших алгоритмів для обходу графа і є основою для багатьох важливих алгоритмів для роботи з графами. Наприклад, ***алгоритм Прима*** (***Prim***) пошуку мінімального остовного дерева або ***алгоритм Дейкстри*** (***Dijkstra***) пошуку найкоротшого шляху з однієї вершини використовують ідеї, подібні з ідеями, використовуваними при пошуку в ширину.

Нехай заданий граф ***G = (V, E)*** і виділена вихідна (source) вершина ***s*** Алгоритм пошуку в ширину систематично обходить всі ребра ***G*** для "відкриття" всіх вершин, досяжних з ***s,*** обчислюючи при цьому відстань (мінімальна кількість ребер) від ***s*** до кожної досяжної з ***s*** вершини. Крім того, в процесі обходу будується "дерево пошуку в ширину" з коренем ***s***, що містить всі досяжні вершини. Для кожної досяжної з ***s*** вершини ***v*** шлях в дереві пошуку в ширину відповідає найкоротшому (тобто такому, що містить найменшу кількість ребер) шляху від ***s*** до ***v*** в ***G***. Алгоритм працює як для орієнтованих, так і для неорієнтованих графів.

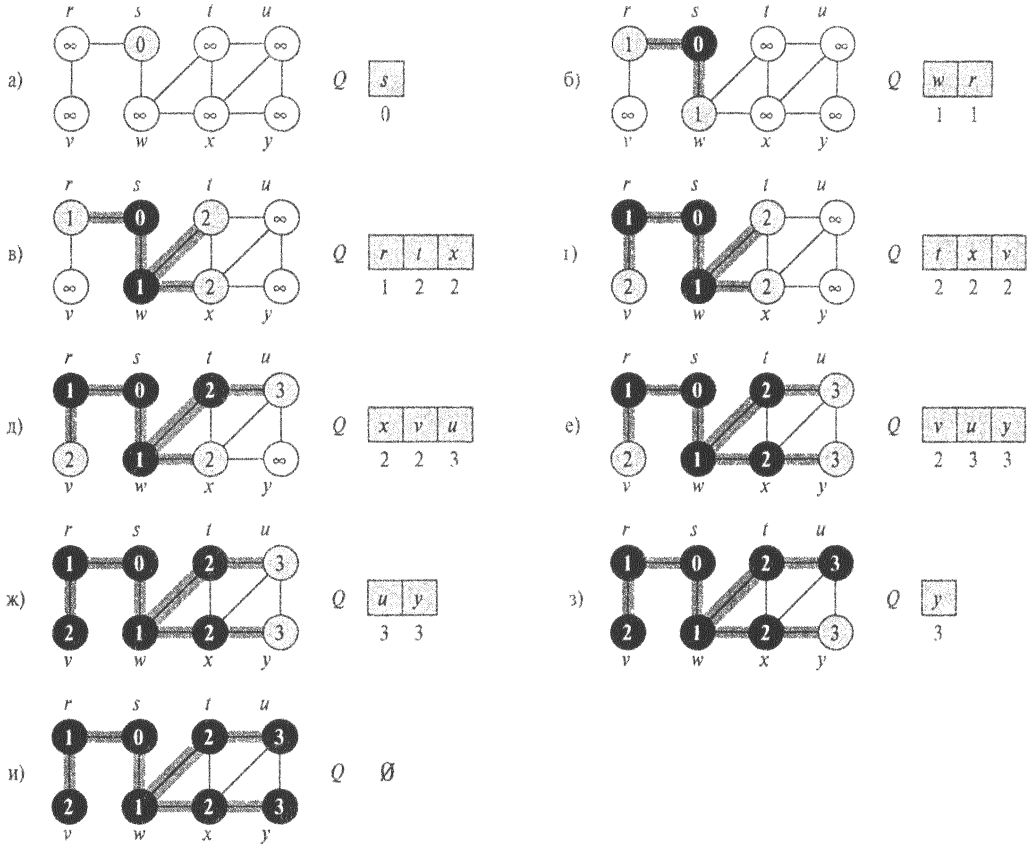


Рис. 9.2. Виконання процедури BFS над неорієнтованим графом

Пошук вшир має таку назву тому, що в процесі обходу ми йдемо вшир, тобто перед тим як приступити до пошуку вершин на відстані ***k+1***, виконується обхід всіх вершин на відстані ***k.***

Для відстеження роботи алгоритму пошук в ширину розфарбовує вершини графа в білий, сірий і чорний кольори. Спочатку всі вершини білі, і пізніше вони можуть стати сірими, а потім чорними. Коли вершина відкривається (discovered) в процесі пошуку, вона забарвлюється. Таким чином, сірі та чорні вершини – це вершини, які вже були відкриті, але алгоритм пошуку в ширину по-різному працює з ними, щоб забезпечити оголошений порядок обходу. Якщо ***(u, v) є E*** і вершина *u* чорного кольору, то вершина *v* або сіра, або чорна, тобто всі вершини, суміжні з чорною, вже відкриті. Сірі вершини можуть мати білих сусідів, представляючи собою границю між відкритими і невідкритими вершинами.

Пошук в ширину будує дерево пошуку в ширину, яке спочатку складається з одного кореня, яким є вихідна вершина ***s***. Якщо в процесі сканування списку суміжності вже відкритої вершини ***u*** відкривається біла вершина ***v***, то вершина ***v*** і ребро ***(u, v)*** додаються в дерево. Кажуть, що ***u*** є попередником (predecessor), або

батьком (parent), ***v*** в дереві пошуку вшир. Оскільки вершина може бути відкрита не більше одного разу, вона має не більше одного з батьків. Взаємовідносини предків і нащадків визначаються в дереві пошуку в ширину як звичайно – якщо ***u*** знаходиться на шляху від кореня ***s*** до вершини ***v***, то ***u*** є предком ***v,*** а ***v*** – нащадком ***u***.

Наведена нижче процедура пошуку в ширину ***BFS*** передбачає, що вхідний граф ***G = (V, E)*** представлений за допомогою списків суміжності. Крім того, підтримуються додаткові структури даних в кожній вершині графа. Колір кожної вершини ***u ϵ V*** зберігається в змінній ***color [u]***, а попередник – у змінній ***π[u]***. Якщо попередника у ***u*** немає (наприклад, якщо ***u = s*** або ***u*** не відкрита), то ***π[u] = NIL***. Відстань від ***s*** до вершини ***u***, що обчислюється алгоритмом, зберігається в полі ***d[u]***. Алгоритм використовує чергу ***Q*** для роботи з множиною сірих вершин:

```
BFS(G, s)
for (для) кожної вершини u ϵ V[G] – s
2               do color[u] ← WHITE
3                       d[u] ← ∞
4                       π[u] ← NIL
5       color[s] ← GRAY
6       d[s] ← 0
7       π[s] ← NIL
8       Q←Ø
9       ENQUEUE(Q, s)
10      while Q ≠ Ø
11              do u ← DEQUEUE(Q)
12                      for (для) каждой v ϵ Adj[u]
13                              do if color [v] = WHITE
14                                      then color[v] ← GRAY
15                                              d[v] ← d[u]+1
16                                              π[u] ← u
17                                              ENQUEUE(Q, v)
18                      color[u] ← BLACK
```

Процедура ***BFS*** працює наступним чином. У рядках 1-4 всі вершини, за винятком вихідної вершини ***s***, фарбуються в білий колір, для кожної вершини u полю ***d[u]*** присвоюється значення ∞, а батьком для кожної вершини встановлюється значення ***nil***. У рядку 5 вихідна вершина s забарвлюється в сірий колір, оскільки вона розглядається як відкрита на початку процедури. У рядку 6 її полю ***d[s]*** присвоюється значення 0, а в рядку 7 її батьком стає nil. У рядках 8-9 створюється порожня черга ***Q***, в яку поміщається один елемент ***s***.

Цикл while в рядках 10-18 виконується до тих пір, поки залишаються сірі вершини (тобто відкриті, але списки суміжності яких ще не переглянуті).

Інваріант даного циклу виглядає наступним чином:

При виконанні перевірки в рядку 10 черга ***Q*** складається з множини сірих вершин.

Перед першою ітерацією єдиною сірою вершиною і єдиною вершиною в черзі ***Q***, є вихідна вершина s. У рядку 11 визначається сіра вершина u в голові черги ***Q***, яка потім видаляється з черги. Цикл ***for*** в рядках 12-17 переглядає всі вершини ***v*** в

списку суміжності u. Якщо вершина ***v*** біла, значить, вона ще не відкрита, і алгоритм відкриває її, виконуючи рядки 14-17. Вершині призначається сірий колір, дистанція ***d[v]*** встановлюється рівною ***d[u] + 1***, а в якості її батька вказується вершина ***u***. Після цього вершина поміщається в хвіст черги ***Q***. Після того як всі вершини зі списку суміжності ***u*** переглянуті, вершині ***u*** присвоюється чорний колір. Інваріант циклу зберігається, так як всі вершини, які фарбуються в сірий колір (рядок 14), вносяться в чергу (рядок 17), а вершина, яка видаляється з черги (рядок 11), забарвлюється в чорний колір (рядок 18).

Результат пошуку в ширину може залежати від порядку перегляду вершин, суміжних з даною вершиною, в рядку 12. Дерево пошуку в ширину може варіюватися, але відстані ***d***, обчислені алгоритмом, не залежать від порядку перегляду.

ПОШУК ВГЛИБ

Пошук в глибину (англ. Depth-first search, DFS) – це рекурсивний алгоритм обходу вершин графа. Стратегія пошуку в глибину, як випливає з її назви, полягає в тому, щоб йти "вглиб" графа, наскільки це можливо. При виконанні пошуку в глибину досліджуються всі ребра, що виходять з вершини, відкритої останньої, і залишає вершину, тільки коли не залишається недосліджених ребер – при цьому відбувається повернення в вершину, з якої була відкрита вершина ***v***. Цей процес триває до тих пір, поки не будуть відкриті всі вершини, досяжні з вихідної. Якщо при цьому залишаються невідкриті вершини, то одна з них вибирається в якості нової вихідної вершини і пошук повторюється вже з неї. Цей процес повторюється до тих пір, поки не будуть відкриті всі вершини.

Як і в разі пошуку в ширину, коли вершина ***v*** відкривається в процесі сканування списку суміжності вже відкритої вершини ***u***, процедура пошуку записує цю подію, встановлюючи поле попередника ***v π[v]*** рівним ***u.***

На відміну від пошуку в ширину, де підграф передування утворює дерево, при пошуку в глибину підграф передування може складатися з декількох дерев, так як пошук може виконуватися з декількох вихідних вершин.

Підграф передування (predecessor subgraph) пошуку в глибину, таким чином, дещо відрізняється від такого для пошуку в ширину. Ми визначаємо його як граф $G_\pi = (V, E_\pi)$, де

$$E_\pi = \{(\pi[v], v): v \in E\ V \text{ і } \pi[v] \neq NIL\}.$$

Підграф передування пошуку в глибину утворює ліс ***пошуку в глибину*** (***depth-first forest***), який складається з декількох дерев ***пошуку в глибину*** (***depth-first trees***). Ребра в E_π називаються ребрами дерева (tree edges).

Як і в процесі виконання пошуку в ширину, вершини графа розфарбовуються в різні кольори, які свідчать про їх стан. Кожна вершина спочатку біла, потім при відкритті (discover) в процесі пошуку вона забарвлюється в сірий колір, і по завершенні (finish), коли її список суміжності повністю просканований, вона стає чорною. Така методика гарантує, що кожна вершина в кінцевому рахунку знаходиться тільки в одному дереві пошуку в глибину, так що дерева не перетинаються.

Крім побудови лісу пошуку в глибину, пошук в глибину також проставляє в вершинах мітки часу (timestamp). Кожна вершина має дві такі мітки – першу ***d[v]***, в якій вказується, коли вершина *v* відкривається (і забарвлюється в сірий колір), і

друга – ***f[v],*** яка фіксує момент, коли пошук завершує сканування списку суміжності вершини *v* і вона стає чорною.

Ці мітки використовуються багатьма алгоритмами і корисні при розгляді поведінки пошуку в глибину.

Наведена нижче процедура DFS записує в поле ***d[u]*** момент, коли вершина ***u*** відкривається, а в поле ***f[i]*** – момент завершення роботи з вершиною ***u***. Ці мітки часу являють собою цілі числа в діапазоні від 1 до 2 |***V***|, оскільки для кожної з |***V***| вершин є тільки одна подія відкриття і одна – завершення. Для кожної вершини ***u***

$$d[u] < f[u].$$

До моменту часу ***d[u]*** вершина має колір white, між ***d[u]*** і ***f[u]*** – колір GRAY, а після ***f[u]*** – колір BLACK.

Далі представлений псевдокод алгоритму пошуку в глибину. Вхідний граф ***G*** може бути як орієнтованим, так і неорієнтованим. Змінна ***time*** – глобальна і використовується для міток часу.

```
DFS(G)
1       for (для) кожної вершини u є V[G]
2               do color[u] ← WHITE
3                       π[u] ← NIL
4       time ← 0
5       for (для) кожної вершини u є V[G]
6               do if color [u] = WHITE
7                       then DFS_Visit(u)
DFS_Visit(u)
1       color [u] ← GRAY
2       time ← time+1
3       d[u] ← time
4       for (для) кожної вершини v є Adj[u]
5               do if color [v] = WHITE
6                       then π[v] ← u
7                               DFS_Visit(v)
8       color[u] ← BLACK
9       f[u] ← time← time+1
```

На рис. 9.3 проілюстровано виконання процедури ***DFS*** над графом, наведеному на рис. 9.1 (2 частина). Ребра, досліджені алгоритмом, або зафарбовані (якщо це ребра дерев), або позначені пунктиром (в іншому випадку). Ребра, які не є ребрами дерев, позначені на малюнку буквами ***B*** (зворотні – back), ***F*** (прямі – forward) і ***C*** (перехресні – cross). У вершинах вказані часові мітки в форматі відкриття/завершення.

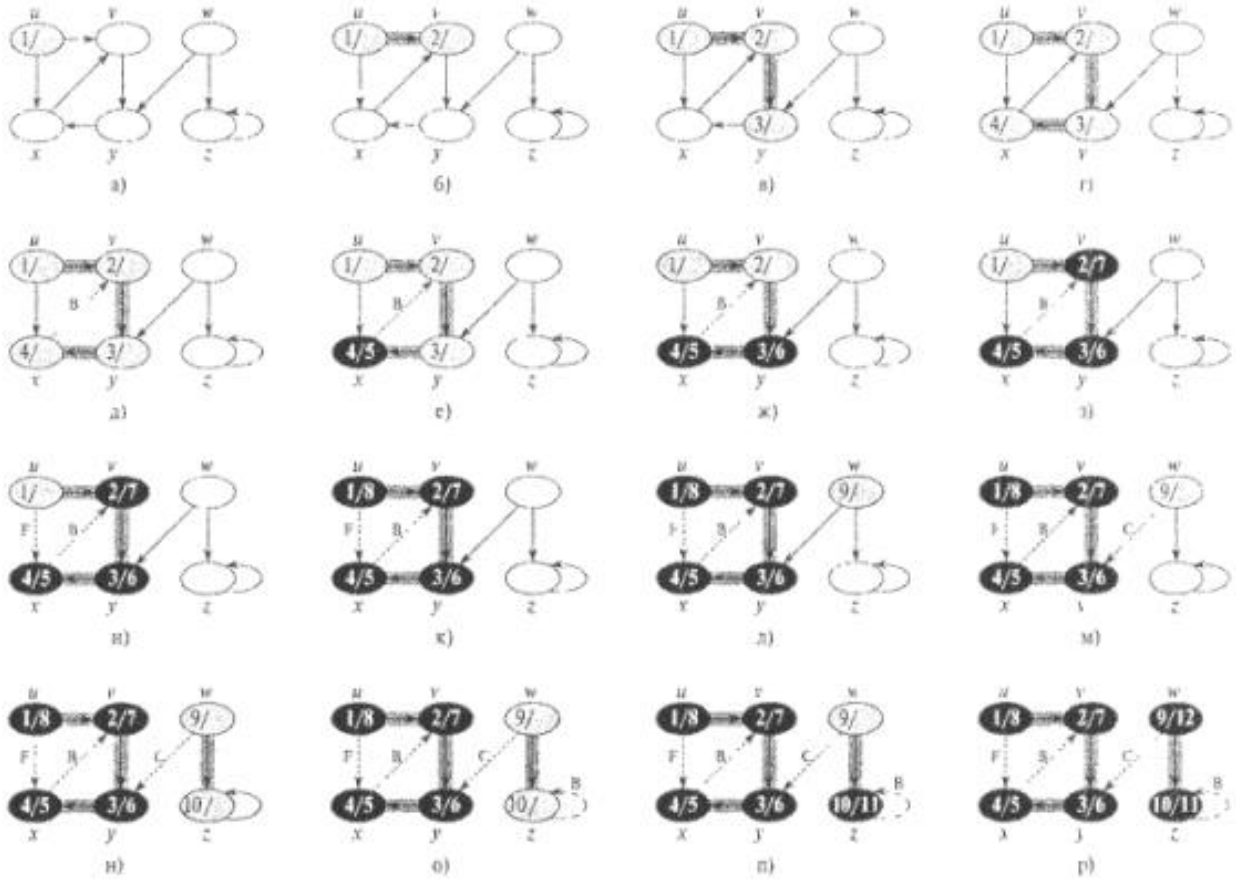


Рис. 9.3. Виконання процедури DFS над неорієнтованим графом

Процедура ***DFS*** працює наступним чином. У рядках 1-3 все вершини фарбуються в білий колір, а їх поля π ініціалізуються значенням ***nil***. У рядку 4 виконується скидання глобального лічильника часу. У рядках 5-7 по черзі перевіряються всі вершини з ***V,*** і коли виявляється біла вершина, вона обробляється за допомогою процедури ***DFS_Visit***. Кожен раз при виклику процедури ***DFS_Visit(u)*** в рядку 7, вершина *u* ***стає*** коренем нового дерева лісу пошуку в глибину. При поверненні з процедури ***DFS*** кожній вершині *u* зіставляються два моменти часу – час відкриття (discovery time) ***d[u]*** і час завершення (finishing time) ***f[i]***.

При кожному виклику ***DFS_Visit(u)*** вершина ***u*** спочатку має білий колір. У рядку 1 вона забарвлюється в сірий колір, в рядку 2 збільшується глобальна змінна time, а в рядку 3 виконується запис нового значення змінної time в поле часу відкриття ***d[u]***. У рядках 4-7 досліджуються всі вершини, суміжні з ***u***, і виконується рекурсивне відвідування білих вершин. При розгляді в рядку 4 вершини ***v∈Adj [u]***, говоримо, що ребро ***(u, v)*** досліджується (explored) пошуком в глибину. І нарешті, після того як будуть досліджені всі ребра, які виходять з ***u***, в рядках 8-9 вершина ***u*** забарвлюється в чорний колір, а в поле ***f[u]*** записується час завершення роботи з нею.

Зауважимо, що результат пошуку в глибину може залежати від порядку, в якому виконується розгляд вершин в рядку 5 процедури ***DFS***, а також від порядку відвідування суміжних вершин в рядку 4 процедури ***DFS_Visit.***

Комп'ютерний практикум № 10

Алгоритм Дейкстри

Мета роботи: ознайомитися з алгоритмом пошуку найкоротших шляхів.

Завдання

Реалізувати алгоритм Дейкстри для знаходження найкоротшої відстані між будь-якими двома вершинами заданого графу.

На вхід подається зважений граф, який заданий текстовим файлом. Перший рядок цього файлу містить два числа – кількість вершин та ребер. Далі йдуть рядки, в кожному з яких представлена інформація про одне орієнтоване ребро. Ребро задається трьома числами: "***u v w***", де ***u*** – початкова вершина ребра, ***v*** – кінцева вершина ребра, ***w*** – довжина (вага) ребра. У файлі можуть бути кратні ребра (тобто такі, що з'єднують одні й ті самі вершини). Індексація вершин починається з 1.

Теоретичні відомості

АЛГОРИТМ ДЕЙКСТРИ

Алгоритм Дейкстри вирішує задачу знаходження найкоротшого шляху з однієї вершини в зваженому орієнтованому графі ***G = (V, E)*** в тому випадку, коли ваги ребер невід'ємні. Для всіх ребер ***(u, v) ϵ E*** виконується нерівність ***w (u, v) ≥ 0***.

В алгоритмі Дейкстри підтримується множина вершин ***X***, для яких вже враховано остаточні ваги найкоротших шляхів до них зі стартової вершини s. У цьому алгоритмі по черзі вибирається вершина ***u*** ϵ ***V[G] – X***, якій на даному етапі відповідає мінімальна оцінка найкоротшого шляху.

```
Dijkstra(G, s)
for (для) кожної вершини v ϵ V[G] do
   A[v] ← ∞; B[v] ← NIL
A[s] ← 0
X ← {s}; v ← s
while X ≠ V[G]:
   for (для) каждого ребра (v, u) ϵ E[G] та u ϵ V[G] – X:
      if A[u]>A[v]+w(v, u) then
         A[u] ← A[v]+w(v, u)
         B[u] ← v
   for (серед) усіх значень вершин з V[G]– X знайти v* з min значенням A[v]
      X ← X + {v *}; v = v*
   return A, B
```

Алгоритм Дейкстри проілюстрований на рис. 10.1. Стартова вершина ***s*** розташована на малюнку зліва від інших вершин. У кожній вершині приведена оцінка найкоротшого шляху до неї, а виділені ребра вказують попередників. Чорним кольором позначені вершини, додані до множини ***X***, а білим – що містяться в ***V – X***. В частині а малюнка проілюстрована ситуація, що склалася безпосередньо перед виконанням першої ітерації циклу while в рядках 5-11.

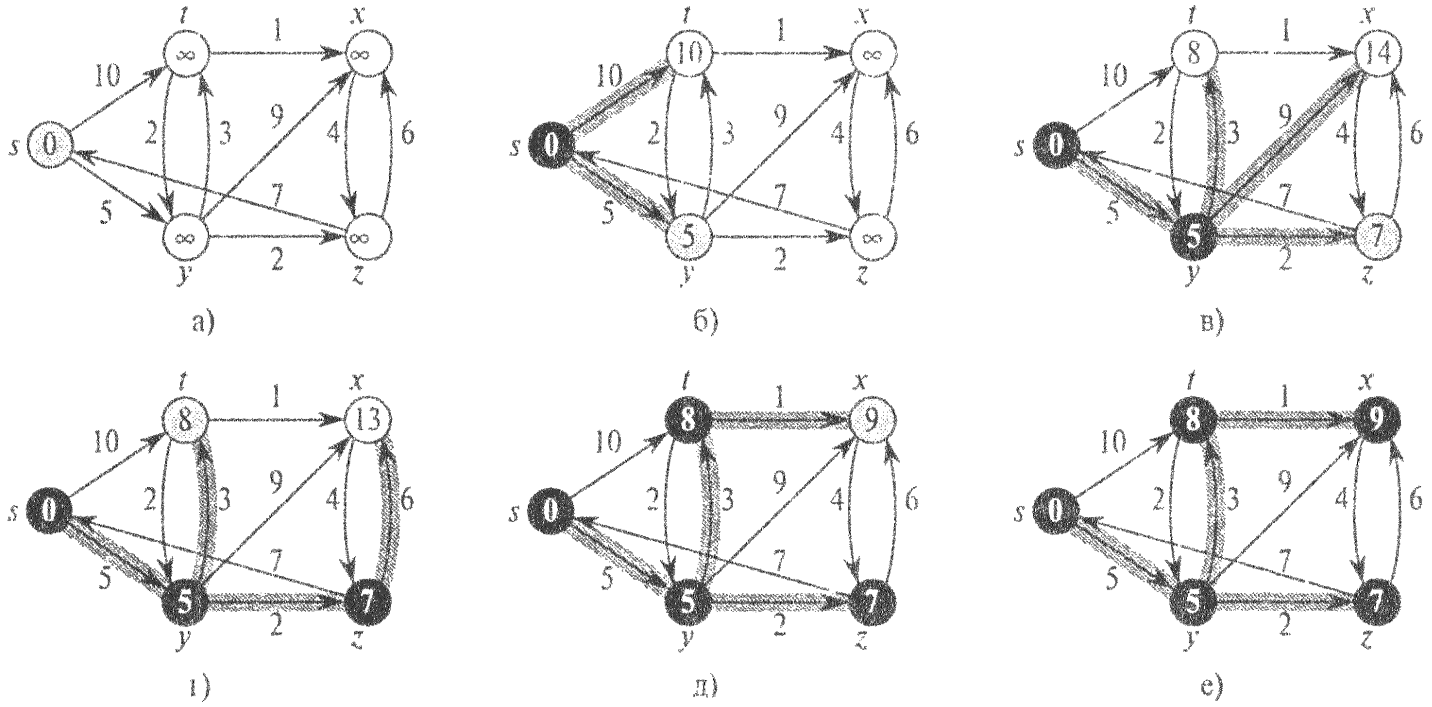


Рис. 10.1. Виконання алгоритму Дейкстри

Виділена сірим кольором вершина має мінімальне значення і обирається в якості вершини ***u*** для наступної ітерації. У частинах б-е зображені ситуації після виконання чергової ітерації циклу while. У кожній з цих частин виділена сірим кольором вершина вибирається в якості вершини ***u***. У частині е наведені кінцеві значення найкоротших шляхів.

Робота розглянутого алгоритму.

У рядках 1-4 проводиться звичайна ініціалізація величин ***A[v]*** і ***B[v]***. Найкоротша відстань від ***s*** до всіх інших вершин ***v*** множини ***V***, яка зберігається у масиві ***A*** =∞. Масив B зберігає вершини попередники у найкоротшому шляху. У в рядку 3 позначається, що найкоротша відстань від ***s*** до самої себе = 0.

А в 4 рядку ініціалізується множина вершин ***X***, що спочатку містить лише стартову вершину ***s*** і вказується що початкова вершина ***v*** – це і є ***s***.

У цьому алгоритмі підтримується інваріант, згідно з яким цикл while виконується до тих пір доки множина ***X*** не буде містити всі вершини графу ***G***.

Серед усіх ребер що виходять з вершини ***v*** у всі вершини які належать множині ***V – X*** якщо поточний найкоротший шлях до вершини ***u*** може бути поліпшений в результаті проходження через вершину ***v***, виконується відповідне оновлення оцінки величини ***A[u]*** і попередника ***B[u]***.

Після цього обирається нова поточна вершина ***v**** яка має мінімальне значення ***A[v]*** та заноситься до множини ***X***.

Оскільки в алгоритмі Дейкстри з множини ***V - S*** для приміщення в множину ***X*** завжди вибирається сама "легка" або "близька" вершина, кажуть, що цей алгоритм дотримується жадібної стратегії.