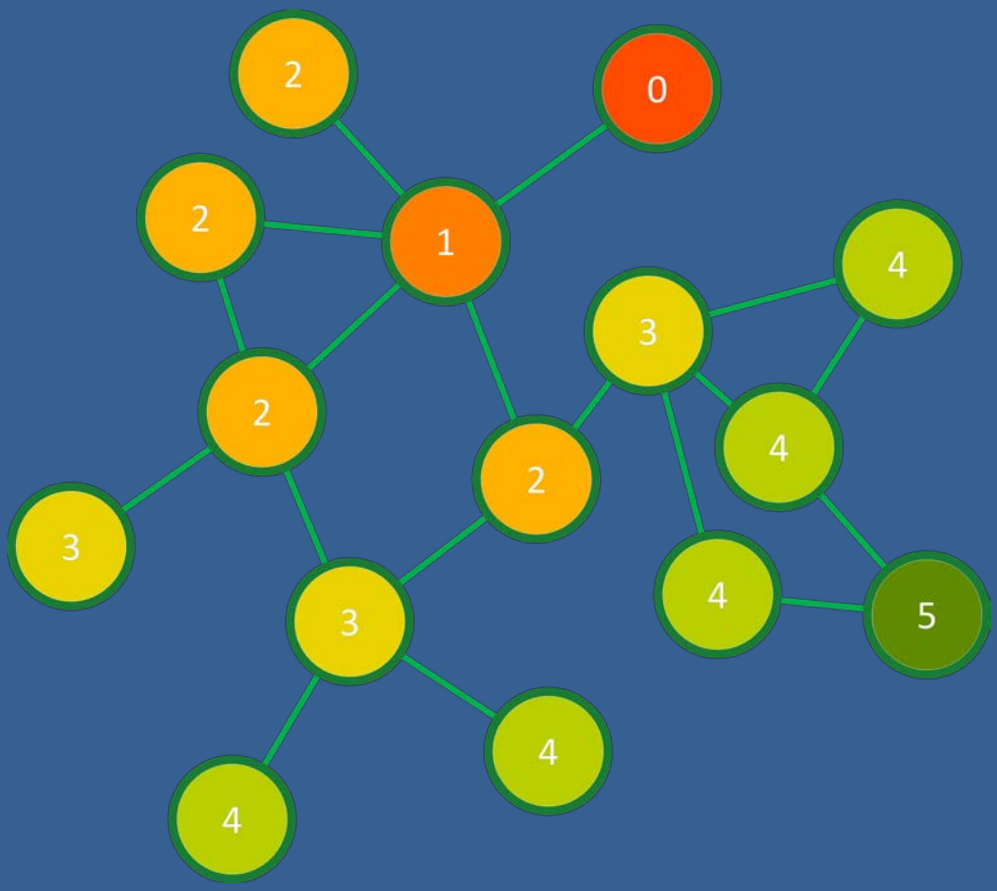
Андрій Креневич

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчальний посібник



КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

2018

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

КРЕНЕВИЧ А.П.

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчальний посібник

КИЇВ 2018

УДК 004.438С(075.8); 519.682

Рецензенти:
доктор фіз.-мат. наук
кандидат техн. наук

Креневич А.П.
Алгоритми та структури даних. Навчальний посібник. – К.: ВПЦ "Київський Університет", 20___. – 777 с.

Посібник призначений для практичного опанування програмування із застосуванням мови Python. Він охоплює основні розділи структурного програмування, що викладаються у вищих навчальних закладах для студентів математичних, природничих та інженерних спеціальностей.

У посібнику у систематизованому вигляді наводяться короткі теоретичні відомості, типові приклади розв'язання задач і задачі для самостійної роботи. Посібник складено з урахуванням досвіду викладання програмування на механіко-математичному факультеті Київського національного університету імені Тараса Шевченка.

Для студентів університетів та викладачів, що проводять практичні заняття з програмування.

ЗМІСТ

ВСТУП

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафів

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва

івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

іва
івафі
фівафіва

РОЗДІЛ 1. РЕКУРЕНТНІ СПІВВІДНОШЕННЯ ТА РЕКУРСІЯ

§1.1. Рекурентні співвідношення

Рекурентні співвідношення мають надзвичайно важливе значення для програмування. Вони використовуються у аналізі алгоритмів, наближених обчисленнях, динамічному програмуванні, тощо. Тому вивчення курсу пропонуємо читачу розпочати саме з цього розділу.

Рекурентне співвідношення першого порядку

Нехай $\{a_k : k \geq 0\}$ деяка послідовність дійсних чисел.

Означення 1.1. Послідовність $\{a_k : k \geq 0\}$ називається заданою **рекурентним співвідношенням першого порядку**, якщо явно задано її перший член a_0, а кожен наступний член a_k цієї послідовності визначається деякою залежністю через її попередній член a_{k-1}, тобто

$$\begin{cases} a_0 = u \\ a_k = f(n, p, a_{k-1}),\ k \geq 1 \end{cases}$$

де u задане (початкове) числове значення, p – деякий сталий параметр або набір параметрів, f – функція, задана аналітично у вигляді арифметичного виразу, що складається з операцій доступних для виконання з допомогою мови програмування (зокрема у нашому випадку Python).

Для прикладу розглянемо послідовність $\{a_k = k! : k \geq 0\}$. Її можна задати рекурентним співвідношенням першого порядку. Дійсно, враховуючи означення факторіалу отримаємо

$$\begin{cases} a_0 = 1 \\ a_k = k a_{k-1},\ k \geq 1 \end{cases}$$

Маючи рекурентне співвідношення можна знайти який завгодно член послідовності. Наприклад, якщо потрібно знайти a_5, то використовуючи рекурентні формули, послідовно від першого члена отримаємо

$$\begin{gathered} a_0 = 1 \\ a_1 = 1 \cdot a_0 = 1 \cdot 1 = 1 \\ a_2 = 2 \cdot a_1 = 2 \cdot 1 = 2 \\ a_3 = 3 \cdot a_2 = 3 \cdot 2 = 6 \\ a_4 = 4 \cdot a_3 = 4 \cdot 6 = 24 \\ a_5 = 5 \cdot a_4 = 5 \cdot 24 = 120 \end{gathered}$$

З точки зору програмування, послідовність задана рекурентним співвідношенням значно зручніша, ніж задана у явному вигляді. Для обчислення членів послідовностей, заданих рекурентними співвідношеннями, використовують цикли.

Нехай послідовність a_k задана рекурентним співвідношенням

$$\begin{cases} a_0 = u \\ a_k = f(n, p, a_{k-1}),\ k \geq 1 \end{cases}$$

Тоді, після виконання коду

```
a = u
for k in range(1, n + 1):
    a = f(k, p, a)
```

у змінній a буде міститися значення елемента a_n послідовності

Приклад 1.1. Для введеного з клавіатури значення n обчислити $n!$

Розв'язок. Як було зазначено раніше послідовність $a_k = k!$ може бути задана рекурентним співвідношенням

$$\begin{cases} a_0 = 1 \\ a_k = k a_{k-1},\ k \geq 1 \end{cases}$$

Тоді, згідно з вищенаведеним алгоритмом, отримаємо

```
n = int(input("n = "))
a = 1                       # a = u
for k in range(1, n+1):
    a = k * a               # a = f(k, p, a)
print ("%d! = %d" % (n, a))# виводимо на екран результат
```

Приклад 1.2. Скласти програму для обчислення елементів послідовності

$$a_k = \frac{x^k}{k!},\ k \geq 0.$$

Розв'язок. Складемо рекурентне співвідношення для заданої послідовності. Легко бачити, що кожен член послідовності a_k є добутком чисел. Враховуючи це, обчислимо частку двох сусідніх членів послідовності. Для $k \geq 1$ отримаємо

$$\frac{a_k}{a_{k-1}} = \frac{x^k}{k!} \cdot \frac{(k-1)!}{x^{k-1}} = \frac{x}{k}$$

Звідки випливає, що для $k \geq 1$

$$a_k = \frac{x}{k} a_{k-1}$$

Отже ми отримали для послідовності a_k рекурентну формулу, у якій кожен член послідовності для всіх $k \geq 1$ визначається через попередній член a_{k-1}. Щоб задати рекурентне співвідношення, залишилося задати перший член a_0. Для цього просто підставимо 0 у вихідну формулу

$$a_0 = \frac{x^0}{0!} = 1.$$

Отже остаточно отримаємо рекурентне співвідношення першого порядку

$$\begin{cases} a_0 = 1 \\ a_k = \frac{x}{k} a_{k-1},\ k \geq 1 \end{cases}$$

Тоді, згідно з вищенаведеним алгоритмом, отримаємо програму

```
n = int(input("n = "))
x = float(input("x = "))
a = 1
for k in range(1, n+1):
    a = x / k * a       # можна так: a *= x / k
print ("a = ", a)       # виводимо на екран результат
```

Зауважимо, що нумерація членів послідовності інколи починається не з 0, а з деякого натурального числа m, тобто $\{a_k : k \geq \mathrm{m}\}$. Припустимо, що рекурентне співвідношення для цієї послідовності має вигляд

$$\begin{cases} a_m = u, \\ a_k = f(k, p, a_{k-1}), k \geq m+1. \end{cases}$$

Тоді для того, щоб отримати a_n, необхідно замінити наведений вище алгоритм на такий

```
a = u
for k in range(m + 1, n + 1):
    a = f(k, p, a)
```

який, отриманий заміною стартового значення у інструкції **range** на значення m + 1**.**

Приклад 1.3. Скласти програму обчислення суми:

$$S_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Розв'язок. Зазначимо, що задане співвідношення має сенс тільки для $n \geq 1$. Складемо рекурентне співвідношення для послідовності $\{S_k : k \geq 1\}$. Помічаємо, що на відміну від попереднього прикладу, кожен член послідовності S_k є сумою елементів вигляду $1/i$, де i змінюється від 1 до k. Отже, для

побудови рекурентного співвідношення знайдемо різницю двох сусідніх членів послідовності S_k. Для $k \geq 2$

$$S_k - S_{k-1} = 1/k$$

Підставляючи у вихідне співвідношення $k = 1$, отримаємо $S_1 = 1$. Отже, рекурентне співвідношення для послідовності S_k матиме вигляд:

$$\begin{cases} S_1 = 1 \\ S_k = S_{k-1} + \dfrac{1}{k},\ k \geq 2 \end{cases}$$

Аналогічно до попереднього прикладу, враховуючи, що нумерація членів послідовності починається з 1, а не з нуля, отримаємо програму.

```
n = int(input("n = "))
S = 1
for k in range(2, n + 1):
    S += 1 / k
print("S = ", S)
```

Приклад 1.4. Скласти програму обчислення суми

$$S_n = \sum_{i=1}^{n} 2^{n-i} i^2,\ n \geq 1.$$

Розв'язок. Розглянемо послідовність

$$\left\{ S_k = \sum_{i=1}^{k} 2^{k-i} i^2 : k \geq 1 \right\}$$

Складемо для неї рекурентне співвідношення. Підставляючи $k = 1$, отримаємо $S_1 = 1$. Щоб отримати вираз для загального члена, розкриємо суму для $k \geq 2$

$$S_k = 2^k \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(k-1)^2}{2^{k-1}} + \frac{k^2}{2^k} \right) =$$
$$= 2 \cdot 2^{k-1} \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(k-1)^2}{2^{k-1}} + \frac{k^2}{2^k} \right) =$$
$$2 \cdot 2^{k-1} \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(k-1)^2}{2^{k-1}} \right) + 2 \cdot 2^{k-1} \frac{k^2}{2^k} = 2 \cdot S_{k-1} + k^2$$

Отже, рекурентне співвідношення для буде мати вигляд

$$\begin{cases} S_1 = 1, \\ S_k = 2 \cdot S_{k-1} + k^2,\ k \geq 2. \end{cases}$$

і відповідно програма

```
n = int(input("n = "))
S = 1
for k in range(2, n + 1):
    S = 2 * S + k ** 2
print("S = ", S)
```

Рекурентні співвідношення старших порядків

Нехай $\{a_k : k \geq 0\}$ деяка послідовність дійсних чисел. m – деяке натуральне число більше за одиницю.

Тоді

Означення 1.2. Послідовність $\{a_k : k \geq 0\}$ називається заданою **рекурентним співвідношенням *m*-го порядку**, якщо

$$\begin{cases} a_0 = u,\ a_1 = v, \ldots, a_{m-1} = w, \\ a_k = f(n, p,\ a_{k-1}, \ldots, a_{k-m},),\ k \geq m \end{cases}$$

де $u, v, \dots, w$ – задані числові сталі, p – деякий сталий параметр або набір параметрів, f – функція, задана аналітично у вигляді арифметичного виразу, що складається з операцій доступних для виконання з допомогою мови програмування.

Найпоширенішим прикладом послідовності заданої рекурентним співвідношенням 2-го порядку є послідовність чисел Фібоначчі. Перші два члени цієї послідовності дорівнюють одиниці, а кожен наступний член є сумою двох попередніх

$$\begin{cases} F_0 = 1,\ F_1 = 1 \\ F_k = F_{k-1} + F_{k-2},\ k \geq 2 \end{cases}$$

Як і у випадку рекурентного співвідношення першого порядку, маючи рекурентне співвідношення можна знайти який завгодно член послідовності.

$$\begin{gathered} F_0 = 1,\ F_1 = 1 \\ F_2 = F_1 + F_0 = 1 + 1 = 2 \\ F_3 = F_2 + F_1 = 2 + 1 = 3 \\ F_4 = F_3 + F_2 = 3 + 2 = 5 \\ F_5 = F_4 + F_3 = 5 + 3 = 8 \\ F_6 = F_5 + F_4 = 5 + 3 = 13 \end{gathered}$$

Для обчислення елементів послідовності, заданої рекурентним співвідношенням вищого порядку, застосовується інший підхід ніж для співвідношень першого порядку.

Алгоритм наведемо на прикладі співвідношення 3-го порядку. Нехай послідовність a_k задана рекурентним співвідношенням

$$\begin{cases} a_0 = u,\ a_1 = v, a_2 = w, \\ a_k = f(k, p,\ a_{k-1},, a_{k-2}, a_{k-3},),\ k \geq 3 \end{cases}$$

Тоді, після виконання коду

```
a3 = u  # a3 - змінна для (k-3)-го члену послідовності
a2 = v  # a2 - змінна для (k-2)-го члену послідовності
a1 = w  # a1 - змінна для (k-1)-го члену послідовності
for k in range(3, n + 1):
    # Обчислення наступного члену
    a = f(k, p, a1, a2, a3)
    # Зміщення змінних для наступних ітерацій
    a3 = a2
    a2 = a3
    a1 = a
```

у змінних a і a1 буде міститися a_n, у змінній a2 – a_{n-1} , а в змінній a3 – a_{n-2}.

Звернемо увагу на той факт, що для обчислення членів послідовності заданої рекурентним співвідношенням першого порядку не потрібно жодних додаткових змінних – лише змінна у якій обчислюється поточний член послідовності. Для рекурентних співвідношень старших порядків, крім змінної, у якій обчислюється поточних член послідовності, необхідні ще додаткові змінні, кількість яких дорівнює порядку рекурентного співвідношення.

Приклад 1.5. Знайти n-й член послідовності Фібоначі

Розв'язок. Як було зазначено раніше послідовність чисел Фібоначчі F_n може бути задана рекурентним співвідношенням

$$\begin{cases} F_0 = 1,\ F_1 = 1 \\ F_k = F_{k-1} + F_{k-2},\ k \geq 2 \end{cases}$$

Оскільки послідовність Фібоначчі задана рекурентним співвідношенням другого порядку, то для того, щоб запрограмувати обчислення її членів, необхідно три змінних. Модифікувавши наведений вище алгоритм для обчислення відповідного члена послідовності заданої рекурентним співвідношенням третього порядку на випадок рекурентного співвідношення другого порядку, отримаємо програму

```
n = int(input("n = "))
F2 = 1
F1 = 1
for k in range(2, n + 1):
    F = F1 + F2
    F2 = F1
    F1 = F
```

```
print("F = ", F)
```

Приклад 1.6. Скласти програму для обчислення визначника порядку n:

$$D_n = \begin{vmatrix} 5 & 3 & 0 & 0 & \dots & 0 & 0 \\ 2 & 5 & 3 & 0 & \dots & 0 & 0 \\ 0 & 2 & 5 & 3 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 2 & 5 \end{vmatrix}.$$

Розв'язок. Легко обчислити, що

$$D_1 = 5;$$
$$D_2 = \begin{vmatrix} 5 & 3 \\ 2 & 5 \end{vmatrix} = 19.$$

Розкладаючи для всіх $k \geq 3$ визначник D_k по першому рядку отримаємо рекурентне співвідношення

$$D_k = 5D_{k-1} - 6D_{k-2}, k \geq 3.$$

Тоді, згідно з вищенаведеним алгоритмом, програма для знаходження n-го члена послідовності D_k буде мати вигляд

```
n = int(input("n = "))
D2 = 5   # 1-й член послідовності
D1 = 19  # 2-й член послідовності
for k in range(3, n + 1):
    D = 5 * D1 - 6 * D2
    D2 = D1
    D1 = D
print("D_%d = %d" % (n, D1))
```

Системи рекурентних співвідношень

Вищенаведена теорія рекурентних співвідношень легко узагальнюється на системи рекурентних співвідношень, якщо вважати, що послідовності у означеннях вище є векторними.

Розглянемо системи рекурентних співвідношень на прикладах

Приклад 1.7. Скласти програму для обчислення суми

$$S_n = \sum_{i=0}^{n} \frac{x^i}{i!}, \ , n \geq 0.$$

Розв'язок. Розглянемо послідовність

$$\left\{ S_k = \sum_{i=1}^{k} \frac{x^i}{i!} : k \geq 0 \right\}$$

Складемо для неї рекурентне співвідношення.

Розкриваючи суму побачимо, що для всіх $k \geq 1$

$$S_k = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots + \frac{x^{k-1}}{(k-1)!} + \frac{x^k}{k!} = S_{k-1} + \frac{x^k}{k!}$$

Отже послідовність S_k визначається рекурентним співвідношенням

$$\begin{cases} S_0 = 1, \\ S_k = S_{k-1} + \dfrac{x^k}{k!}, \ k \geq 1 \end{cases}$$

Позначимо

$$a_k := \frac{x^k}{k!}, \ k \geq 0.$$

У прикладі 1.2 для цієї послідовності було отримано рекурентне співвідношення. Тоді для вихідної послідовності S_k система рекурентних співвідношень матиме вигляд

$$\begin{cases} S_0 = 1,\ a_0 = 1 \\ a_k = \dfrac{x}{k} a_{k-1},\ k \geq 1, \\ S_k = S_{k-1} + a_k,\ k \geq 1. \end{cases}$$

Отже, програма для знаходження S_n буде мати вигляд

```
n = int(input("n = "))
x = float(input("x = "))
a = 1
S = 1
for k in range(1, n + 1):
    a = x / k * a
    S = S + a
print ("S = ", S)
```

Приклад 1.8. Скласти програму для обчислення суми

$$S_n = \sum_{k=0}^{n} a^k b^{n-k}$$

Розв'язок. Як і у попередніх приклад розглянемо послідовність

$$\left\{ S_k = \sum_{i=1}^{k} a^i b^{k-i} : k \geq 0 \right\}$$

Рекурентне співвідношення можемо побудувати двома способами.

Спосіб 1. Очевидно, що $S_0 = 1$. Розкриваючи суму і групуючи доданки аналогічно до прикладу 1.4, отримаємо

$$\begin{cases} S_0 = 1, \\ S_k = b \cdot S_{k-1} + a^k, \qquad k \geq 1. \end{cases}$$

Введемо позначення $x_k = a^k$. Запишемо для послідовності $\{x_k : k \geq 0\}$ рекурентне співвідношення:

$$\begin{cases} x_0 = 1, \\ x_k = a \cdot x_{k-1}, \qquad k \geq 1. \end{cases}$$

Таким чином, отримаємо систему рекурентних співвідношень

$$\begin{cases} S_1 = x_1 = 1, \\ x_k = a \cdot x_{k-1}, \qquad k \geq 1. \\ S_k = b \cdot S_{k-1} + x_k, \end{cases}$$

Спосіб 2. Легко бачити, що послідовність S_k можна зобразити у вигляді

$$S_k = \frac{a^{k-1} - b^{k-1}}{a - b}, \qquad k \geq 1$$

Тоді система рекурентних співвідношень буде мати вигляд

$$\begin{cases} x_1 = a, y_1 = b, \\ x_k = a \cdot x_{k-1}, \\ y_k = b \cdot y_{k-1}, \qquad k \geq 1. \\ S_k = \dfrac{x_k - y_k}{a - b}, \end{cases}$$

Програма для знаходження n-го члена послідовності S_k, заданого рекурентним співвідношенням, котре отримано першим способом, має вигляд:

```
n = int(input("n = "))
a = float(input("a = "))
b = float(input("b = "))

S = x = 1
for k in range(1, n + 1):
```

```
    x = a * x
    S = b * S + x

print(S)
```

Приклад 1.9. Обчислити суму

$$S_n = \sum_{i=1}^{n} \frac{a_i}{2^i},$$

де $a_1 = a_2 = a_3 = 1$, $a_k = a_{k-1} + a_{k-3}$, $k \geq 4$.

Розв'язок. Звернемо увагу на те, що послідовність a_k задана рекурентним співвідношенням третього порядку. Введемо допоміжну послідовність $b_k = 2^k, k \geq 0$, для якої рекурентне співвідношення буде мати вигляд $b_1 = 1, b_k = 2b_{k-1}, k \geq 1$.

Тоді, поєднуючи алгоритми для визначення відповідних членів послідовностей, заданих рекурентними співвідношеннями першого і третього порядків, отримаємо програму

```
n = int(input("n = "))
a1 = a2 = a3 = 1  # Одночасна ініціалізація кількох
                  # змінних одним значенням
b = 1
S = 1 / 2
# обчислення перших трьох членів послідовності S
for k in range(1, min(4, n + 1)):
    b = 2 * b
    S = S + 1 / b
for k in range(4, n + 1):
    b = 2 * b
    a = a1 + a3
    S = S + a / b
    a3 = a2
    a2 = a1
    a1 = a
print(S)
```

Приклад 1.10. Обчислити суму

$$S_n = \sum_{k=0}^{n} \frac{a_k}{1 + b_k},$$

де

$$\begin{cases} a_0 = 1, \\ a_k = a_{k-1} b_{k-1}, \end{cases} \quad \begin{cases} b_0 = 1, \\ b_k = a_{k-1} + b_{k-1}, \end{cases} \quad k \geq 1.$$

Розв'язок. Послідовності $\{a_k\}$ і $\{b_k\}$ задані рекурентним співвідношеннями першого порядку, проте залежність перехресна. Використаємо по одній допоміжній змінній для кожної з послідовностей.

Тоді, програма для знаходження S_n:

```
n = int(input("n = "))

S = 0.5
a = 1
b = 1
for k in range (1, n+1):
    a_k = a * b  # допоміжна змінна для a_k
    b_k = a + b  # допоміжна змінна для b_k
    a = a_k
    b = b_k
    S = S + a / (1 + b)

print(S)
```

Приклад 1.11. Обчислити добуток

$$P_n = \prod_{k=0}^{n} \frac{a_k}{3^k},$$

де $a_0 = a_1 = 1, a_2 = 3, a_k = a_{k-3} + \frac{a_{k-2}}{2^{k-1}}, k \geq 3$.

Розв'язок. Послідовність $\{a_k\}$ задана рекурентним співвідношенням третього порядку. Тоді добуток P_n обчислюється за допомогою рекурентного співвідношення

$$\begin{cases} P_2 = 1/9, \\ P_k = P_{k-1} \cdot a_k / z_k, \qquad k \geq 3, \end{cases}$$

де z_k – k-й степінь числа 3, визначений рекурентним співвідношенням

$$\begin{cases} z_2 = 9, \\ z_k = 3z_{k-1}, \qquad k \geq 3. \end{cases}$$

Передбачивши змінну t для обчислення членів послідовності $\{t_k = 2^{k-1} : k \geq 3\}$, отримаємо програму

```
n = int(input("n = "))

P = 1.0 / 9.0
z = 9
t = 2
a2 = a3 = 1
a1 = 3
for k in range(3, n + 1):
    z = z * 3
    t = t * 2
    a = a3 + a2 / t
    a3 = a2
    a2 = a1
    a1 = a
    P = P * a / z

print(P)
```

Відшукання членів послідовності, що задовольняють визначену умову

Досі ми будували програми, що знаходять значення члену послідовності за його номером. Проте, часто постає задача, коли потрібно знайти найперший член послідовності, що задовольняє певну умову. У такому разі цикл **for** по діапазону значень заміняється циклом з умовою **while**. Умова у цьому циклі є запереченням до умови, яка визначає коли потрібно припинити обчислення членів послідовності.

Розглянемо приклади

Приклад 1.12. Для довільного натурального $N \geq 2$ знайти найменше число вигляду 3^k, де k – натуральне, таке, що $3^k \geq N$.

Розв'язок. Розглянемо послідовність $a_k = 3^k, k \geq 0$. Легко бачити, що її можна задати рекурентним співвідношенням першого порядку

$$\begin{cases} a_0 = 1, \\ a_k = 3a_{k-1}, \qquad k \geq 1. \end{cases}$$

Отже, враховуючи, що послідовність a_k строго зростаюча, щоб виконати завдання задачі необхідно обчислювати члени послідовності a_k в циклі використовуючи вищенаведене рекурентне співвідношення, доки не знайдемо перший такий, що $a_k \geq N$. Відповідно умова у циклі, буде запереченням до $a_k \geq N$, тобто $a_k < N$. Далі очевидним чином маємо програму

```
N = int(input("N = "))

a = 1
while a < N:
    a = a * 3

print(a)
```

Приклад 1.13. Послідовність задана рекурентним співвідношенням

$$\begin{cases} x_0 = 1,\ x_1 = 0,\ x_2 = 1, \\ x_k = x_{k-1} + 2x_{k-2} + x_{k-3},\ k \geq 3. \end{cases}$$

Створити програму для знаходження найбільшого члена цієї послідовності разом з його номером, який не перевищує число a.

Розв'язок. Запишемо кілька перших членів заданої послідовності

$$1, 0, 1, 2, 4, 9, 19, 41, 88$$

Звернемо увагу на те, що ця послідовність є зростаючою. Тоді, для того, щоб знайти найбільший член цієї послідовності, що не перевищує задане число a, необхідно обчислювати члени цієї послідовності, доки не знайдемо перший такий член, що буде більшим за задане число a. Тоді, член послідовності, що вимагається умовою задачі – елемент послідовності, що передує знайденому.

Наприклад, якщо $a = 30$, то перший член послідовності, що більший за число 30 є 41, а відповідно шуканий згідно з умовою задачі член послідовності – той який йому передує, тобто 19.

Нехай `x, x1, x2, x3` – змінні, що використовуються згідно до алгоритму для обчислення членів послідовності заданої вищенаведеним рекурентним співвідношенням третього порядку. Нагадаємо, що тоді у змінних `x, x1` будуть знаходитися поточні члени послідовності x_n. Тоді, умова циклу буде `x1 <= a`.

Таким чином, маємо програму

```
a = int(input("a = "))
x3 = 1
x2 = 0
x1 = 1
k = 2             # Номер поточного члену послідовності
while x1 <= a:    # Поки поточний член послідовності менший за a
    k += 1
    x = x1 + 2 * x2 + x3
    x3 = x2
    x2 = x1
    x1 = x

print ("x(%d) = %d <= %d = a" % (k - 1 , x2, a))
```

Наведемо результат виконання програми для числа $a = 30$.

```
a = 30
x(6) = 19 <= 30 = a
```

Завдання для самостійної роботи

1.1.

§1.2. Наближені обчислення границь послідовностей

Одне з важливих призначень рекурентних співвідношень це апарат для наближених обчислень границь послідовностей та значень аналітичних функцій.

Нехай задано послідовність $\{y_k : k \geq 0\}$, така, що $y_k \to y, k \to \infty$.

Означення 1.3. Під наближеним з точністю ε значенням границі послідовності y_k будемо розуміти такий член y_n послідовності, що виконується співвідношення

$$|y_n - y_{n-1}| < \varepsilon$$

Вищенаведене означення не є строгим з точки зору чисельних методів. У загальному випадку математичний апарат наближеного обчислення границь послідовностей та значень алгебраїчних функцій перебуває поза межами цього посібника і вимагає від читача додаткових знань з теорії чисельних методів [2].

Отже, згідно з наведеним вище означенням, для того щоб знайти значення границі послідовності потрібно обчислювати елементи послідовності доки виконується умова

$$|y_n - y_{n-1}| \geq \varepsilon$$

Розглянемо застосування зазначеного підходу на прикладах.

Приклад 1.14. Скласти програму наближеного обчислення золотого перетину c, використовуючи:

а) границю

$$\Phi = \lim_{k\to\infty} \frac{F_k}{F_{k-1}}$$

де F_k – послідовність чисел Фібоначчі;

б) ланцюговий дріб

$$\Phi = 1 + \frac{1}{1 + \frac{1}{1+\ddots}}.$$

Розв'язок. Золотий перетин це число

$$\Phi \approx 1{,}6180339887..$$

яке має багато унікальних властивостей, причому не лише математичних. Це число можна зустріти як у різноманітних сферах діяльності людини (наприклад, у мистецтві, архітектурі, культурі) так і у оточуючому нас світі, зокрема фізиці, біології тощо. Для того, щоб переконатися у тому, що наш алгоритм працює правильно, скористаємося однією з властивостей золотого перетину, а саме:

$$\Phi - 1 = \frac{1}{\Phi}.$$

а) Розглянемо послідовність

$$\Phi_k = \frac{F_k}{F_{k-1}}, k \geq 1.$$

Знайдемо рекурентне співвідношення для c_k. Очевидно, що $c_1 = 1$, далі для $k \geq 2$ отримаємо

$$\Phi_k = \frac{F_k}{F_{k-1}} = \frac{F_{k-1} + F_{k-2}}{F_{k-1}} = 1 + \frac{1}{\frac{F_{k-1}}{F_{k-2}}} = 1 + \frac{1}{\Phi_{k-1}}.$$

б) Розглянемо послідовність

$$\Phi_k = 1 + \frac{1}{1 + \frac{1}{1+\ddots + \frac{1}{1}}},$$

що містить $k-1$ риску дробу. Очевидно, що для цієї послідовності рекурентне співвідношення буде таким же, як у пункті а).

Напишемо програму, що знаходить наближене з точністю ε значення границі послідовності Φ_k. Використаємо змінну `current` для обчислення поточного члену послідовності Φ_k і змінну `prev`, у якій будемо запам'ятовувати попередній член Φ_{k-1} цієї послідовності.

Тоді програма має вигляд

```
eps = 0.0000000001  # точність
prev = 0    # попередній член послідовності
current = 1 # поточний член послідовності

while abs(current - prev) >= eps:
    prev = current
    current = 1 + 1 / current

print("Φ =", current)  # Виводимо значення золотого перетину

# Перевірка результату згідно з властивостями
print("Φ - 1 =", current - 1)
print("1 / Φ =", 1 / current)
```

Наведемо результат виконання програми.

```
Φ = 1.618033988738303
Φ - 1 = 0.6180339887383031
```

```
1 / Φ = 0.6180339887543225
```

Приклад 1.15. За допомогою розкладу функції e^x в ряд Тейлора

$$y(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^k}{k!} + \cdots$$

обчислити з точністю $\varepsilon > 0$ її значення для заданого значення x.

Розв'язок. Позначимо загальний член вищенаведеного ряду через a_k, а його часткову суму

$$S_k = \sum_{i=0}^{k} \frac{x^i}{i!},$$

Очевидно, що $S_k \to e^x$, $k \to \infty$.

У прикладі 1.7 було отримано, що послідовність S_k визначається системою рекурентних співвідношень

$$\begin{cases} S_0 = 1,\ a_0 = 1 \\ a_k = \dfrac{x}{k} a_{k-1}, \quad k \geq 1, \\ S_k = S_{k-1} + a_k,\ k \geq 1. \end{cases}$$

Відповідно до означення, наведеного вище і з огляду на рекурентне співвідношення, під наближеним значенням границі послідовності S_k будемо розуміти такий член S_n, що виконується співвідношення

$$|S_n - S_{n-1}| = |a_n| < \varepsilon$$

Отже, програма буде мати вигляд

```
eps = 0.0000000001  # точність

x = float(input("x = "))
a = 1
S = 1
k = 0
while abs(a) >= eps:
    n += 1
    a = x / k * a
    S = S + a

print("exp(%f) = %f" % (x, S))
```

Звернемо увагу на те, що на відміну від цикла **for**, цикл **while** не має вбудованого лічильника. Тому, оскільки нам необхідно враховувати у формулі номер члена послідовності, то ми задали змінну n, яка відіграє роль лічильника.

Наведемо результат виконання програми.

```
x = 1.0
exp(1.000000) = 2.718282
```

Завдання для самостійної роботи

1.2.

§1.3. Рекурсія

Рекурсія – спосіб визначення об'єкту (або методу) попереднім описом одного чи кількох його базових випадків, з наступним описом на їхній основі загального правила побудови об'єкту.

З рекурсією ми часто зустрічаємося у оточуючому житті: рекурсивні зображення, структура рослин та кристалів, рекурсивні розповіді, вірші або жарти.

Рекурсивна функція – метод визначення функції, при якому функція прямо або неявно викликає сама себе.

Розглянемо вищенаведене означення на прикладі.

Як ми знаємо послідовність чисел Фібоначчі визначається рекурентним співвідношенням другого порядку

$$\begin{cases} F_0 = 1,\ F_1 = 1, \\ F_n = F_{n-1} + F_{n-2},\ n \geq 2. \end{cases}$$

Отже, якщо покласти, що функція $F(n)$ визначає n-те число послідовності Фібоначчі, то з вищенаведеного рекурентного співвідношення отримаємо, що послідовність Фібоначчі може бути визначена рекурсивною функцією:

$$\begin{cases} F(0) = 1,\ F(1) = 1, \\ F(n) = F(n-1) + F(n-2),\ n \geq 2. \end{cases}$$

Початкові значення $F(0) = 1, F(1) = 1$, дуже важливі при визначенні рекурсивної функції. Якби їх не було, то рекурсія б стала нескінченною! Тому, описуючи рекурсивну функцію завжди треба переконуватися, що для всіх допустимих значень аргументів виклик рекурсивної функції завершиться, тобто що рекурсія буде скінченною.

Можна звернути увагу, на те, що на відміну від обчислення елементів послідовності заданої рекурентним співвідношенням, де обчислення відбувається він тривіального (початкового) елементу до шуканого, рекурсивна функція починає обчислення від шуканого.

Кількість вкладених викликів функції називається **глибиною рекурсії**. Наприклад, глибина рекурсії при знаходженні $F(5)$ буде 4.

Опишемо стандартний алгоритм опису рекурсивної функції. Структурно рекурсивна функція на верхньому рівні завжди є розгалуженням, що складається з двох або більше альтернатив, з яких

- принаймні одна є термінальною (припинення рекурсії);
- принаймні одна є рекурсивною (тобто здійснює виклик себе з іншими аргументами).

```
def recursive_func(args):
    if terminal_case:       # Термінальна гілка
       ...
       return trivial_value # Тривіальне значення
    else:                   # Рекурсивна гілка
       ...
       # Виклик функції з іншими аргументами
       return expr(recursive_func(new_args))
```

Для прикладу опишемо рекурсивну підпрограму для обчислення чисел Фібоначчі. У програмі цю функцію будемо позначати `Fib(n)`.

```
def Fib(n):
    if n == 0 or n == 1:              # Термінальна гілка
       return 1                       # Тривіальне значення
    else:                             # Рекурсивна гілка
       return Fib(n - 1) + Fib(n - 2) # Рекурсивний виклик
# Виклик рекурсивної функції
n = int(input("n = "))
print("Fib(%d) = %d" % (n, Fib(n)))
```

Рекурсія використовується, коли можна виділити **самоподібність** задачі. Розглянемо інший класичний приклад, який використовується у навчальній літературі для пояснення рекурсії.

Приклад 0.1. Описати рекурсивну функцію для знаходження факторіала натурального числа.

Розв'язок. Очевидно, що функцію $F(n) = n!$ можна задати у такому рекурсивному вигляді

$$\begin{cases} F(0) = 1, \\ F(n) = nF(n-1),\ n \geq 1. \end{cases}$$

Тоді програма, разом з рекурсивною функцією (у програмі будемо позначати її `Fact(n)`) буде мати такий вигляд

```
def Fact(n):
    if n == 0 :               # Термінальна гілка
       return 1               # Тривіальне значення
    else:                     # Рекурсивна гілка
       return n * Fact(n - 1) # Рекурсивний виклик
```

```
# Виклик рекурсивної функції
n = int(input("n = "))
print("%d! = %d" % (n, Fact(n)))
```

Приклад 0.2. Описати рекурсивну функцію перевірки рядка на симетричність.

Розв'язок. Спочатку опишемо термінальні випадки. Для цього проаналізуємо найпростіші рядки. Очевидно, що порожній рядок або рядок, що складається з одного символа (наприклад, `""`, `"a"`) є симетричним.

Тепер побудуємо рекурсивний виклик. Очевидно, що якщо перший і останній символи рядка є однаковими і рядок без них є симетричним, то вихідний рядок є симетричним.

```
# Рекурсивна функція перевірки рядка S на симетричність
def is_symetric(s):
    if len(s) <= 1:     # якщо рядок S порожній або
                        # складається з одного символа
        return True     # то він симетричний
    else:
        # cond1 - умова, що 1-й і останній символи однакові
        cond1 = s[0] == s[len(s)-1]
        # cond2 - умова, що рядок без першого і останнього
        # символа є симетричним - рекурсивний виклик
        cond2 = is_symetric(s[1 : len(s) - 1])
        # рядок симетричний, якщо обидві умови істині
        return cond1 and cond2
# Виклик рекурсивної функції для введеного з клавіатури рядка
S = input("S = ")
sym = is_symetric(S)

if sym:
    print("Заданий рядок є симетричним")
else:
    print("Заданий рядок не є симетричним")
```

Питання про використання рекурсії дуже суперечливе і неоднозначне. З одного боку, рекурсивна форма як правило значно простіша і наглядніша. З іншого боку, рекомендується уникати рекурсивних алгоритмів, що можуть приводити до занадто великої глибини рекурсії, особливо у випадках, коли такі алгоритми мають очевидну реалізацію у вигляді звичайного циклічного алгоритму. З огляду на це, вищенаведений рекурсивний алгоритм визначення факторіала є прикладом скоріше того, як не треба застосовувати рекурсію.

Теоретично будь-яку рекурсивну функцію можна замінити циклічним алгоритмом (можливо з застосуванням стеку).

Завдання для самостійної роботи

1.3.

РОЗДІЛ 2. СКЛАДНІСТЬ АЛГОРИТМІВ

§2.1. Алгоритм та оцінка його складності

У процесі вирішення різноманітних прикладних задач часто зіштовхуються з проблемою вибору алгоритму. Це часто викликає серйозні труднощі, адже правильний вибір оптимального алгоритму має вирішальне значення. Дійсно, часто правильний з аналітичної точки зору алгоритм дає незадовільний час виконання програми. Інший же алгоритм не можливо реалізувати через вимоги раціонального використання ресурсів комп'ютера (пам'яті, жорсткого диску. тощо).

Для того, щоб зрозуміти, чи підходить алгоритм для розв'язання певної задачі проводять його аналіз.

Алгоритм та його аналіз

Отже, що таке алгоритм і у чому полягає його аналіз?

Означення 2.1. **Алгоритмом** називається набір інструкцій, який описує порядок дій виконавця, щоб розв'язати задачу за скінченну кількість дій (за скінченний час).

Алгоритм може бути виражений багатьма способами. Наприклад, він може бути записаний різними мовами у термінах близьких до тих, що ми використовуємо у звичайному житті. Проте нас будуть цікавити алгоритми, які точно записані використовуючи математичний формалізм. Такі алгоритми записуються з використанням деякої мови програмування.

Означення 2.2. Програмою або **реалізацією алгоритму** називається запис алгоритму за допомогою мови програмування для його виконання комп'ютером.

Після того, як алгоритм реалізований, проводять його аналіз. У результаті цього можна отримати відповідь на питання чи підходить алгоритм для розв'язання поставленої задачі. Аналіз полягає визначенні та аналізі таких критеріїв як:

1. час роботи програми, як функцію від вхідних даних;
2. загальну кількість пам'яті, що необхідна для даних програми;
3. загальний об'єм програмного коду;
4. чи програма розв'язує коректно (правильно) поставлену задачу;
5. чи стресо-стійка програма, тобто як добре буде поводити себе програма з некоректними вхідними даними.
6. комплексність програми, тобто чи легко програму читати, розуміти та модифікувати.

І якщо останні чотири пункти частіше за все покладаються на зовнішні аналізатори (інженери-тестувальники, автоматичні системи тестування, рецензування коду), то перші два цілковито знаходяться у зоні відповідальності програміста, який створює програму. А відтак у цьому курсі зосередимося саме на них.

Цей розділ здебільшого зосереджений на першому пункті вищенаведеного переліку, а саме на оцінці часу виконання програми.

Час виконання (running time)

Припустимо для розв'язання певної задачі побудований алгоритм. Потрібно оцінити на скільки оптимальним (швидким) є цей алгоритм. Виникає низка цілком закономірних питань:

- Як це зробити, тобто оцінити швидкодію цього алгоритму?
- У яких одиницях цю швидкодію вимірювати?
- Яка врешті буде складність цього алгоритму?

Звичайно більшість читачів, які вперше зіштовхнулися з цією проблематикою, скажуть, що швидкодію алгоритму можна оцінити як «час виконання програми» побудованої за цим алгоритмом і будуть правими. Проте, швидше за все, більшість з них буде вкладати в термін «час» кількість секунд, хвилин чи годин за які виконується програма написана за цим алгоритмом. А ось тут не все так однозначно. Таке означення не є коректним, оскільки воно лише частково характеризує швидкодію алгоритму. Дійсно, ні для кого не секрет, що швидкодія різних комп'ютерів різна і якщо на одному комп'ютері програма буде виконуватися, припустимо 10 секунд, то на комп'ютері, що в 10 разів швидший – 1 секунду. Крім цього, час роботи програми побудованої за алгоритмом буде залежати від мови програмування на якій її написано.

Тому цілком логічним є те, що для оцінки складності алгоритму необхідно використовувати критерій, що не залежить від потужності ЕОМ або мови програмування на якій реалізовано алгоритм.

Робота будь-якої програми, що виконується комп'ютером складається з елементарних операцій, які об'єднуються у блоки для утворення інструкцій мови програмування. До елементарних операцій, у цьому курсі віднесемо операції з набору

- звернення до об'єкту в пам'яті;
- присвоєння;
- елементарні арифметичні операції (додавання, віднімання, множення, ділення, ділення без остачі та остача від ділення) ;
- операції порівняння;
- булеві оператори;
- виклик функції/методу;
- повернення результату функцією;
- звернення за індексом до елементу списку (масиву);
- звернення за ключем до елементу словника.

Як ви можете здогадуватися, час, який витрачає комп'ютер на різні типи інструкцій різний. Наприклад, час інструкцій звернення до об'єкту в пам'яті та присвоєння значення змінній може суттєво відрізнятися. Проте, для дослідження питань цього курсу, нам буде достатнього розглядати спрощену модель комп'ютера, вважаючи, що всі елементарні операції виконуються за однаковий час τ, і без обмежень загальності, можемо вважати, що

$$\tau = 1.$$

Таким чином

Означення 2.3. **Часом виконання програми** (eng. running time, time complexity) будемо називати кількість елементарних операцій, які виконує комп'ютер під час виконання програми.

Очевидно, що ця кількість операцій може залежати від вхідних даних (inputs) задачі. Дійсно, наприклад очевидно, що для знаходження визначника матриці розміром 3 треба здійснити значно більше операцій, ніж для знаходження визначника розміром 2.

Час виконання програми будемо позначати

$$T(n)$$

де n – розмір вхідних даних.

Розглянемо приклади.

Приклад 2.1. Дано вектор заданої величини n. Оцінимо час виконання програми знаходження розміру цього вектора.

Очевидно, що розмір вектора наперед заданий (і зберігається разом з вектором), тому для його визначення достатньо однієї операції – звернення до пам'яті. Таким чином:

$$T(n) = 1$$

У такому разі кажуть, що алгоритм виконується за сталий час.

Приклад 2.2. Знайдемо кількість операцій алгоритму, що обчислює суму компонент заданого вектора розмірності n

$$a = (a_1, \dots, a_n)$$

Функція, що розв'язує цю задачу буде мати вигляд

Лістинг 2.1.

```
1   def sumV(a, n):
2       result = a[0]
3       i = 1
4       while i < n:
5           result += a[i]
6           i+=1
7       return result
```

Визначимо час виконання програми. Для цього створимо таблицю рядки якої будуть відповідати рядкам програми

Рядок	Час
2	3
3	2
4	$3 \times n$
5	$5 \times (n-1)$
6	$4 \times (n-1)$
7	2

Зауваження, тут і надалі ми будемо вважати, що операції `+=`, `-=` і подібні виконуються за 4 операції, оскільки їх можна інтерпретувати як

```
i = i + 1
```

При цьому число `1` є літералом, що зберігається у пам'яті. Від так доступ до нього вимагає одну елементарну операцію.

Таким чином час виконання програми буде

$$T(n) = 2 + 3 + 3n + 5(n-1) + 4(n-1) + 2 = 12n - 2$$

Як бачимо «складність» задачі безпосередньо пов'язана із розміром вектора (вхідні дані). Причому цей час виконання є лінійною функцією від розміру вхідних даних. У такому разі кажуть, що алгоритм має лінійну складність, або програма виконується за лінійний час.

Приклад 2.3. Знайдемо кількість операцій алгоритму, що обчислює суму компонент квадратної матриці розмірності n.

Для розв'язання цієї задачі можна запропонувати такий алгоритм

Лістинг 2.2.

```
1   def sumM(A, n):
2       result = 0
3       i = 0
4       j = 0
5       while i < n:
6           while j < n:
7               result += A[i][j]
8               j += 1
9           j = 0
10          i += 1
11      return result
```

Аналогічно попередньому прикладу побудуємо таблицю кількості операцій кожного рядка

Рядок	Час
2	2
3	2
4	2
5	$3 \times (n+1)$
6	$(3 \times (n+1)) \times (n+1)$
7	$((4+2) \times n) \times n$
8	$(3 \times n) \times n$
9	$2 \times n$
10	$4 \times n$
11	1

Отже, підсумовуючи значення другого стовпчика, отримаємо, що

$$T(n) = 12n^2 + \ldots$$

Час роботи цього алгоритму називають **поліноміальним**, оскільки він оцінюється як поліном від розміру вхідних даних.

Приклад 2.4. Оцінимо час роботи алгоритму рекурсивного знаходження F_n - n-го члена послідовності Фібоначчі.

Така підпрограма буде мати вигляд

Лістинг 2.3.

```
1   def fib(n):
2       if n <= 1:
3           return 1
4       else:
5           return fib(n - 1) + fib(n - 2)
```

Знову побудуємо таблицю

Рядок	Час	
	$n \leq 1$	$n > 1$
2	3	3
3	2	–
4	–	–
5	–	$1 + 4 \times 2 + 1 + T(n-1) + T(n-2)$

Звідки маємо, що

$$T(0) = 5$$
$$T(1) = 5$$

а, для $n \geq 2$ отримаємо, що складність $T(n)$ описаного алгоритму визначається як рекурентне співвідношення

$$T(n) = T(n-1) + T(n-2) + 10.$$

Звідки маємо, що

$$T(n) \geq F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} \approx 2^{0.7n}$$

Такий час роботи називають **експоненціальним**. І кажуть, що алгоритм працює за **експоненціальний час**.

Приклад 2.5. Оцінимо час роботи алгоритму додавання у стовпчик двох натуральних чисел n та m.

Час роботи такого алгоритму буде пропорційним кількості цифр у «довшому числі». Кількість цифр у числі n не перевищує величину $\log_{10} n$, а у числі m – відповідно величину $\log_{10} m$. Отже час роботи такого алгоритму буде пропорційним такому

$$T(n, m) = \log_{10} \max\{n, m\}$$

Такий час роботи називають **логарифмічним** і відповідно кажуть, що алгоритм працює за **логарифмічний час**.

Приклад 2.6. Визначимо час виконання програми, для знаходження часткової суми ряду

$$\sum_{i=0}^{\infty} x^i$$

Розглянемо послідовність часткових сум цього ряду

$$S_n = \sum_{i=0}^{n} x^i, n \geq 0.$$

Напишемо програму найпростішим способом, тобто таку, що використовує рекурентне співвідношення

$$S_0 = 0$$
$$S_n = S_{n-1} + x^n, n \geq 1$$

Лістинг 2.4.

```
1   def S(x, n):
2       sum = 0
3       i = 0
4       while i <= n:
5           prod = 1
6           j = 0
7           while j < i:
8               prod *= x
9               j += 1
10          sum += prod
11          i += 1
12      return sum
```

та підрахуємо кількість операцій у кожному рядку програми:

Рядок	Час
2	2
3	2
4	$3(n+2)$
5	$2(n+1)$
6	$2(n+1)$
7	$4\sum_{i=0}^{n}(i+1)$
8	$4\sum_{i=0}^{n} i$
9	$4\sum_{i=0}^{n} i$
10	$4(n+1)$
11	$4(n+1)$
12	2

Як відомо

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Тоді підсумовуючи, отримаємо час виконання програми

$$T(n) = \frac{11}{2}n^2 + \frac{47}{2}n + 24$$

Тепер напишемо програму, що вирішує поставлену вище задачу, проте скористаємося алгоритмом, що базується на рекурентному співвідношенні, що отримується зі схеми Горнера

$$S_0 = 0$$
$$S_n = S_{n-1} * x + 1, n \geq 1$$

Тоді програма буде мати вигляд

Лістинг 2.5.

```
1   def S2(x, n):
2       sum = 0
3       i = 0
4       while i <= n:
5           sum = sum * x + 1
6           i += 1
7       return sum
```

і відповідна їй таблиця кількості операцій

Рядок	Час
2	2
3	2
4	$3(n+2)$
5	$6(n+1)$
6	$4(n+1)$
7	2
$\boldsymbol{T(n)}$	$13n+22$

Як бачимо, однаково поставлені задачі можуть бути розв'язані по різному і відповідні їхні алгоритми можуть використовувати різну кількість операцій.

Приклад 2.7. Необхідно написати програму піднесення дійсного числа до цілого степеня та оцінити її складність.

Стандартний алгоритм побудований на елементарному рекурентному співвідношенні має вигляд

Лістинг 2.6.

```
1   def pow(x, n):
2       result = 1
3       i = 0
4       while i <= n:
5           result = result * x
6           i += 1
7       return result
```

і, очевидно, має лінійний порядок складності n. Проте виникає цілком логічне запитання, чи не можна прискорити цей алгоритм? Не складно помітити, що функцію піднесення до степеня можна визначити рекурсивним чином, а саме

$$\mathrm{pow}(x,n)=\begin{cases}1, & n=0\\ \mathrm{pow}^2\left(x,\frac{n}{2}\right), & n-\text{парне}\\ x\cdot\mathrm{pow}^2\left(x,\frac{n-1}{2}\right), & n-\text{непарне}\end{cases}$$

або, що те ж саме

$$\mathrm{pow}(x,n)=\begin{cases}1, & n=0\\ \mathrm{pow}^2\left(x,\frac{n}{2}\right), & n-\text{парне}\\ x\cdot\mathrm{pow}\left(x^2,\frac{n-1}{2}\right), & n-\text{непарне}\end{cases}$$

Лістинг 2.7.

```
1   def pow(x, n):
2       if n == 0:
3           return 1
4       elif n % 2 == 0:
5           return pow(x * x, n / 2)
6       else:
7           return x * pow(x * x, (n - 1) / 2)
```

Складемо таблицю для підрахунку часу виконання програми

Рядок	Час		
	$n=0$	$n>0$ – парне	$n>0$ – непарне
2	3	3	3
3	2	–	–
4	–	5	5

5	–	$10 + T(n/2)$	–
7	–	–	$14 + T((n-1)/2)$
$\boldsymbol{T(n)}$	5	$18 + T(n/2)$	$22 + T((n-1)/2)$

$$T(n) = \begin{cases} 5, & n = 0 \\ 18 + T(n/2), & n - \text{парне} \\ 22 + T((n-1)/2), & n - \text{непарне} \end{cases}$$

Спробуємо знайти явний вигляд отриманого рекурентного співвідношення. Припустимо, що $n = 2^k$, для деякого $k > 0$. Тоді для парного n:

$$n/2 = 2^{k-1}$$

Таким чином,

$$T(n) = T(2^k) = 18 + T(2^{k-1}) =$$
$$= 18 + 18 + T(2^{k-2}) = \cdots = 18j + T(2^{k-j}).$$

Підстановка закінчиться тоді, коли $k = j$. Отже

$$T(2^k) = 18k + T(1) = 18k + 20 + T(0) = 18k + 25.$$

Оскільки $n = 2^k$, то $k = \log_2 n$. І тоді час виконання

$$T(n) = 18 \log_2 n + 25$$

Аналогічно, для непарного n:

$$n + 1 = 2^k$$
$$T(n) = T(2^k - 1) = 22 + T(2^{k-1} - 1) =$$
$$= 22 + 22 + T(2^{k-2} - 1) = \cdots = 22j + T(2^{k-j} - 1).$$

Підстановка завершується при $k = j$

$$T(2^k - 1) = 20k + 5$$

або

$$T(n) = 22 \log_2 (n + 1) + 5$$

Таким чином бачимо, що складність цього алгоритму має логарифмічний порядок.

Найкращий, найгірший та середній час виконання

У попередньому пункті у всіх прикладах знайдений час виконання програм залежав лише від розміру вхідних даних. Проте, для багатьох програм час виконання програми часто залежить не стільки від розміру вхідних даних, скільки від самих цих даних. Щоб у цьому переконатися, розглянемо такий приклад.

Приклад 2.8. Визначити чи заданий елемент s міститься у списку a, що містить n елементів.

Оскільки ми нічого не знаємо заздалегідь про елементи списку, то реалізуємо алгоритм послідовного пошуку:

Лістинг 2.8.

```
1   def find(a, n, s):
2       i = 0
3       while i < n:
4           if a[i] == s:
5               return True
6           i += 1
7       return False
```

Функція послідовно перебирає усі елементи списку і якщо зустрічає шуканий елемент, то її виконання переривається. Відповідно, і її час виконання залежить не лише від розміру вхідних даних, але і від того, які дані подаються на вхід у функцію. Так, наприклад, якщо список першим елементом містить шуканий елемент, то цикл виконає своє тіло рівно один раз. Якщо ж список взагалі не містить шуканого елементу або він є останнім елементом у списку, то тіло циклу виконається всі n разів. У

першому випадку говорять про час виконання **у найкращому випадку**, у другому – **у найгіршому випадку**.

Означення 2.4. **Часом виконання програми у найгіршому випадку** (eng. worst-case running time, worst-case time complexity) будемо називати найбільшу кількість елементарних операцій, які виконує комп'ютер під час виконання програми для довільних вхідних даних розміру n.

Означення 2.5. **Часом виконання програми у найкращому випадку** (eng. best-case running time, best-case time complexity) будемо називати найменшу кількість елементарних операцій, які виконує комп'ютер під час виконання програми яка досягається для деякого набору вхідних даних розміру n.

Зобразимо таблицю для найкращого та найгіршого випадку нашого прикладу

Рядок	Час	
	у найкращому випадку	у найгіршому випадку
2	2	2
3	3	$3(n+1)$
4	5	$5n$
5	2	–
6	–	$4n$
7	–	2
$\boldsymbol{T(n)}$	12	$12n+7$

Очевидно, що згадані вище випадки є крайніми та не завжди можуть об'єктивно відображати інформацію про швидкодію алгоритму на практиці для конкретних наборів вхідних даних. Тому, крім згаданих випадків, часто розглядають третій – **час виконання в середньому**.

Означення 2.6. **Часом виконання програми в середньому** (eng. average-case running time, average-case time complexity) будемо називати усереднену кількість елементарних операцій, які виконує комп'ютер під час виконання програми для всіх наборів вхідних даних розміру n.

Очевидно, що час виконання в середньому час залежить від імовірнісного розподілу вхідних даних і, очевидно, може бути визначеним, якщо ці ймовірнісні характеристики відомі. На практиці час виконання у середньому знайти значно складніше, аніж час виконання у найгіршому випадку, враховуючи математичну складність такої задачі. Тому в основному будемо використовувати час виконання у найгіршому випадку, як характеристику часової складності алгоритму.

Повернемося до розгляду прикладу. Фактично задача визначення часу виконання у середньому у цьому випадку рівнозначна визначенню середньої кількості ітерацій циклу. Припустимо для спрощення, що список складається з різних натуральних чисел з діапазону $[0, n-1]$, а шукане число належить діапазону $[0, n]$. Зроблене припущення означає, що шукане число входить у список не більше ніж один раз, причому ймовірність входження шуканого числа на кожній ітерації циклу є однаковою. Тоді очевидно, що в середньому, кількість ітерацій циклу буде $\frac{(n+1)}{2}$. Доповнимо вищенаведену таблицю колонкою з середнім часом виконання

Рядок	Час виконання		
	у найкращому випадку	у найгіршому випадку	у середньому
2	2	2	2
3	3	$3(n+1)$	$\frac{3(n+1)}{2}$
4	5	$5n$	$\frac{5(n+1)}{2}$
5	2	–	–
6	–	$4n$	$\frac{4(n+1)}{2}$
7	–	2	2
$\boldsymbol{T(n)}$	12	$12n+7$	$6n+10$

Зауважимо, що для іншого набору вхідних даних середній час виконання може суттєво відрізнятися від наведеного вище.

У подальшому, якщо конкретно не зазначено який тип часу виконання розглядається, то будемо вважати, що мається на увазі час виконання у найгіршому випадку.

Завдання для самостійної роботи

2.1. Визначте час виконання фрагментів програм заданих нижче

a)
```
i = 0
while i < n:
    k += 1
    i += 1
```

b)
```
i = 1
while i < n:
   k += 1
   i = i * 2
```

c)
```
i = n - 1
while i != 0:
   k += 1
   i = i / 2
```

d)
```
i = 0
while i < n:
   if i % 2 == 0:
       k += 1
   i += 1
```

e)
```
i = 0
while i < n:
    j = 0
    while j < n:
        k += 1
        j += 1
    i += 1
```

f)
```
i = 0
while i < n:
    j = i
    while j < n:
        k += 1
        j += 1
    i += 1
```

g)
```
i = 0
while i < n:
    j = 0
    while j < i * i:
        k += 1
        j += 1
    i += 1
```

h)
```
i = 0
while i < n:
    j = n
    while j != 0:
        k += 1
        j /= 3
    i += 1
```

2.2. Визначте час виконання програми у явному вигляді, якщо для нього відоме рекурентне співвідношення

a) $T(n) = \begin{cases} 1, & n = 0; \\ T(n-1) + 1, & n \geq 1. \end{cases}$

b) $T(n) = \begin{cases} 1, & n \leq a, a > 0; \\ T(n-a) + 1, & n > a. \end{cases}$

c) $T(n) = \begin{cases} 1, & n = 0; \\ 2T(n-1) + 1, & n > 1. \end{cases}$

d) $T(n) = \begin{cases} 1, & n = 0; \\ 2T(n-1) + n, & n > 1. \end{cases}$

e) $T(n) = \begin{cases} 1, & n = 1; \\ T(n/2) + 1, & n \geq 2. \end{cases}$

f) $T(n) = \begin{cases} 1, & n = 1; \\ 2T(n/2) + 1, & n \geq 2. \end{cases}$

g) $T(n) = \begin{cases} 1, & n = 1; \\ 2T(n/2) + n, & n \geq 2. \end{cases}$

§2.2. Асимптотична оцінка складності алгоритмів

Спробуємо зрозуміти, чому так важливо оцінювати часову складність алгоритмів. Припустимо, що ми розглядаємо два алгоритми: A і B, для вирішення заданої задачі. Крім того, скажімо, ми зробили ретельний аналіз часів роботи кожного з алгоритмів і визначили їх як $T_A(n)$ та $T_B(n)$ відповідно, де n – розмір вхідних даних. Тоді досить просто порівняти дві функції $T_A(n)$ та $T_B(n)$, щоб визначити, який алгоритм кращий! Очевидно, що якщо для деякого відомого входу n_0 має місце нерівність $T_A(n_0) \leq T_B(n_0)$, то алгоритм A є кращим за B для цього набору вхідних даних. Проте, якщо ми не знаємо наперед нічого про вхідні дані, то сказати який алгоритм буде кращим не завжди просто.

Давайте розглянемо такий приклад. Припустимо, що величина входу деякої задачі є n. Припустимо, що три студенти по різному розв'язали цю задачу.

Нехай алгоритм першого використовує

$$T_1(n) = 345n^3 + 123n^2 + 98n + 15$$

операцій, алгоритм другого

$$T_2(n) = 12n^4 + 1$$

операцій, а алгоритм третього

$$T_3(n) = 3 \cdot 2^n$$

операцій. Який алгоритм кращий? У випадку, якщо розмір входу буде зовсім невеликий, наприклад $n = 4$, то останній алгоритм буде виконуватися за найменшу кількість операцій. Проте вже при $n = 20$ кращим буде другий алгоритм, а от, якщо n буде великим, наприклад 10000, то другий алгоритм буде використовувати значно більше операцій ніж перший, а останній взагалі можна вважати нескінченним.

Тут слід зауважити, що, як правило нікого не цікавить швидкодія алгоритму при невеликих значенням вхідних даних – для вхідних даних малого розміру час виконання будь-якого алгоритму є задовільним з практичної точки зору. Тому при оцінці швидкодії алгоритмів цікавить випадок саме великого розміру вхідних даних. А отже, взагалі кажучи, якщо нічого наперед не відомо про розмір вхідних даних, то алгоритм першого студента можна вважати найкращим.

Під час оцінки швидкодії алгоритму при великих значеннях розміру вхідних даних, як правило цікавить не точна кількість операцій, яку здійснює програма, а порядок цієї кількості операцій. Дійсно, ви можете не знати точно як реалізують на комп'ютері арифметичні операції, умовні оператори тощо. Наприклад, для додавання двох великих цілих чисел може використовуватися не одна операція процесора, а три (якщо розрядність числа більша, ніж розрядність операційної системи). Тому при оцінці швидкодії програми:

1) У формулі кількості операцій, враховують лише доданок, що зростає найшвидше.
2) Сталий множник при цьому доданку встановлюють рівними 1.

Отриману величину називають **порядком складності алгоритму**.

Таким чином алгоритм першого студента має складність порядку n^3, другого n^4, а третього 2^n.

Отже, відповідь на питання, поставлене на початку цього пункту полягає у намаганні передбачити як зростає час виконання програми з ростом розміру вхідних даних. Наприклад, якщо розмір вхідних даних задачі збільшився у 10 рази. У скільки разів тоді повільніше буде працювати програма?

Для оцінки складності програм, залежно від вхідних даних, використовують O, Ω та Θ символіку яка є формалізованим обґрунтуванням порядку складності алгоритму. Основне їхнє призначення це «грубо» оцінити час виконання алгоритму, а також наскільки швидко зростає час роботи алгоритму зі збільшенням розміру вхідних даних. Фактично тут піде мова про **асимптотичну поведінку** функцій часу виконання, що залежать від вхідних даних алгоритму.

O – символіка

Означення та приклади

Нехай є дві функції $f, g: \mathbb{N} \to \mathbb{R}_+$, (які будемо інтерпретувати як час роботи двох різних алгоритмів на різних довжинах вхідних даних).

Означення 2.7. Кажуть, що $f = O(g)$, якщо $\exists C > 0$ та $\exists n_0 \geq 0$, що $\forall n \geq n_0$: $f(n) \leq Cg(n)$.

У такому разі також кажуть, що «f зростає не швидше ніж g».

Приклад 2.9. Розглянемо функцію $f(n) = 8n + 128$. Очевидно, що $f(n) \geq 0$ для всіх натуральних n. Покажемо, що $f = O(n^2)$. Згідно з означенням, нам треба знайти таке натуральне n_0 і таку сталу $C > 0$, що для всіх $n \geq n_0$

$$8n + 128 \leq Cn^2$$

Не має значення величина сталої – головне чи вона існує. Припустимо, що $C = 1$

$$8n + 128 \leq n^2 \Rightarrow n^2 - 8n - 128 \geq 0$$
$$\Rightarrow (n - 16)(n + 8) \geq 0$$

Очевидно, що остання нерівність виконується при $n \geq 16$. Таким чином, $\exists C = 1$ та $\exists n_0 = 16$, що для всіх $n \geq n_0$

$$8n + 128 \leq n^2$$

Отже,

$$f(n) = O(n^2)$$

Очевидно, що існує безліч значень таких пар C та n_0.

Приклад 2.10.

$$345n^3 + 123n^2 + 98n + 15 = O(n^3)$$

Зауваження. У цьому курсі будемо позначати $\log_2 n =: \log n$.

Приклад 2.11.

$$n \log n = O(n^2)$$

Зауваження. З того, що $f_1 = O(g)$ і $f_2 = O(g)$ **не випливає**, що

$$f_1 = f_2$$

Дійсно, розглянемо функції $f_1 = n^2$ і $f_2 = n$. легко показати, що $f_1 = O(n^2)$ та $f_2 = O(n^2)$

Властивості

Визначення асимптотичної поведінки функції часто буває дуже клопіткою роботою, якщо користуватися лише означенням. Тому, як правило при оцінці асимптотики функції використовують властивості, деякі з яких наведено нижче.

Будемо вважати, що всі функції, що використовуються нижче є додатнозначними функціями натурального аргументу.

Теорема 2.1. Нехай $f_1 = O(g_1)$ і $f_2 = O(g_2)$, тоді

$$f_1 + f_2 = O(\max(g_1, g_2)).$$

Теорема 2.2. Нехай $f = f_1 + f_2$, причому

$$\lim_{n\to\infty} \frac{f_2(n)}{f_1(n)} = L,$$

де L деяка стала. Тоді

$$f = O(f_1).$$

Теорема 2.3. Нехай $f_1 = O(g_1)$ і $f_2 = O(g_2)$, тоді

$$f_1 \times f_2 = O(g_1 \times g_2).$$

Теорема 2.4. Нехай $f_1 = O(g_1)$, тоді для будь-якої функції g_2

$$f_1 \times g_2 = O(g_1 \times g_2).$$

Теорема 2.5. (**Транзитивність**) Нехай $f = O(g)$ і $g = O(h)$. Тоді

$$f = O(h).$$

З доведенням вищенаведених теорем ви можете ознайомитися у підручнику [1, Chapter 3].

Правила визначення асимптотичної поведінки функції

Властивості, наведені вище дозволяють сформулювати прості правила для визначення асимптотичної поведінки функції.

1) Мультиплікативні константи можна опускати, тобто для будь-якої додатної сталої C

$$Cn^3 = O(n^3);$$

2) Нехай $b \geq a$ тоді

$$n^a = O(n^b);$$

3) Нехай $a > 1$. Тоді для будь-якого $k \geq 1$,

$$n^k = O(a^n);$$

4) Для будь-якого $k \geq 1$

$$\log^k n = O(n);$$

5) Якщо функція є сумою кількох функцій, то її асимптотична поведінка визначається доданком, що зростає найшвидше. Наприклад

$$345n^3 + 123n^2 + 98n + 15 = O(n^3).$$

Поширені асимптотичні складності

У таблиці нижче наведені найпоширеніші асимптотичні складності алгоритмів.

Складність	Коментар	Приклади
$O(1)$	Сталий час роботи не залежно від розміру задачі	Пошук у хеш-таблиці
$O(\log\log n)$	Дуже повільне зростання необхідного часу	Очікуваний час роботи інтерполюючого пошуку n елементів
$O(\log n)$	Логарифмічне зростання — подвоєння розміру задачі збільшує час роботи на сталу величину	Швидке обчислення x^n; двійковий пошук у відсортованому масиві з n елементів
$O(n)$	Лінійне зростання — подвоєння розміру задачі подвоїть і необхідний час	Додавання/віднімання чисел з n цифр; лінійний пошук в масиві з n елементів
$O(n\log n)$	Лінеаритмічне зростання — подвоєння розміру задачі збільшить необхідний час трохи більше ніж вдвічі	Сортування злиттям або купою масиву з n елементів.
$O(n^2)$	Квадратичне зростання — подвоєння розміру задачі вчетверо збільшує необхідний час	Елементарні алгоритми сортування масивів з n елементів; Лінійний пошук у квадратній матриці розмірності n.
$O(n^3)$	Кубічне зростання — подвоєння розміру задачі збільшує необхідний час у вісім разів	Звичайне множення матриць
$O(a^n)$	Експоненціальне зростання — збільшення розміру задачі на 1 призводить до a-кратного збільшення необхідного часу; подвоєння розміру задачі підносить необхідний час у квадрат	Деякі задачі комівояжера; Алгоритми пошуку повним перебором

Ω – символіка

Крім вищенаведеної «великого О» застосовуються інші типи асимптотик, про які буде розказано далі.

Означення та приклади

Означення 2.8. Кажуть, що $f = \Omega(g)$, якщо $g = O(f)$.

У такому разі також кажуть, що «f зростає не повільніше ніж g».

Приклад 2.12. Покажемо, що

$$5n^2 - 64n + 256 = \Omega(n^2).$$

Відповідно до означення, нам треба знайти таке натуральне n_0 і таку сталу $C > 0$, що для всіх $n \geq n_0$

$$5n^2 - 64n + 256 \geq Cn^2.$$

Виберемо $C = 1$. Тоді

$$\begin{gathered} 5n^2 - 64n + 256 \geq n^2 \\ \Rightarrow 4n^2 - 64n + 256 \geq 0 \\ \Rightarrow 4(n-8)^2 \geq 0 \end{gathered}$$

Оскільки $(n-8)^2 \geq 0$ для всіх $n \geq 0$, отримуємо, що $n_0 = 1$, що і доводить твердження прикладу.

Приклад 2.13. Використовуючи правила, що наведені у попередньому пункті, можна легко бачити, що

$$n^2 = \Omega(n\log^3 n).$$

Ω-оцінка використовується коли потрібно оцінити нижню межу швидкодії алгоритму. Такі оцінки часто використовуються для того, щоб переконатися, що отриманий алгоритм є оптимальним.

Наприклад, нехай у нас є масив з n елементів (які можна порівнювати). Відомим є твердження, що для того, щоб відсортувати такий масив необхідно $\Omega(n \log n)$. Отже, це означає, що не можливо відсортувати такий масив швидше, наприклад за лінійний час.

Θ - символіка

Означення 2.9. Кажуть, що $f = \Theta(g)$, якщо $f = O(g)$ та $g = O(f)$.

У такому разі також кажуть, що «f та g мають однаковий порядок росту».

Приклад 2.14.

$$345n^3 + 123n^2 + 98n + 15 = \Theta(n^3)$$
$$\log n = \Theta(\log_{10} n^3)$$

Зауваження. Досить часто, під час асимптотичної оцінки складності алгоритмів, записують O-оцінку, маючи на увазі Θ-оцінку. Тому, у подальшому, без додаткових обговорень, будемо користуватися O-оцінкою навіть для випадків, якщо має місце Θ-оцінка.

Асимптотичний аналіз алгоритмів

У попередньому параграфі ми навчилися обчислювати час роботи алгоритмів. Ця задача була досить клопіткою навіть у випадку використання спрощеної моделі комп'ютера у якій ми вважали, що кожна елементарна операція виконується за однакову одиницю часу. Проте, для оцінки складності алгоритму у більшості випадків досить оцінки її асимптотичної складності. А відтак потреба у таких детальних обчисленнях кількості операцій відпадає.

Оцінити асимптотичну складність алгоритмів можна використовуючи методи обчислення часу виконання програми та використовуючи правила наведені у цій темі. Здебільшого визначення асимптотичної складності алгоритму переважно зводиться до аналізу циклів і рекурсивних викликів підпрограм.

Як правило, найбільший інтерес щодо дослідження алгоритму з практичної точки зору складає оцінка часу його роботи у найгіршому випадку. Це аргументовано ти що:

1) це дає оцінку швидкодії алгоритму для довільних вхідних даних визначеного розміру;
2) з практичної точки зору, «погані» вхідні дані трапляються досить часто;
3) час роботи в середньому є досить близьким до часу роботи у найгіршому випадку.

Розглянемо ще раз приклад 2.6 та оцінимо асимптотичну складність кожного з алгоритмів для запропонованих там. Розширимо таблиці визначання часу виконання колонкою асимптотичної оцінки для кожного з алгоритмів.

Отже, для першого алгоритму таблиця буде мати вигляд

Рядок	Час	Асимптотична оцінка «велике O»
2	2	$O(1)$
3	2	$O(1)$
4	$3(n+2)$	$O(n)$
5	$2(n+1)$	$O(n)$
6	$2(n+1)$	$O(n)$
7	$4\sum_{i=0}^{n}(i+1) = 4\frac{(n+2)(n+1)}{2}$	$O(n^2)$
8	$4\sum_{i=0}^{n} i = 4\frac{n(n+1)}{2}$	$O(n^2)$
9	$4\sum_{i=0}^{n} i = 4\frac{n(n+1)}{2}$	$O(n^2)$
10	$4(n+1)$	$O(n)$
11	$4(n+1)$	$O(n)$
12	2	$O(1)$

Отже, використовуючи правила наведені вище, можемо переконатися, що загальна асимптотика вищенаведеного алгоритму є $O(n^2)$.

Таблиця складності для програми написаної другим способом

Рядок	Час	Асимптотична оцінка

2	2	$O(1)$
3	2	$O(1)$
4	$3(n+2)$	$O(n)$
5	$6(n+1)$	$O(n)$
6	$4(n+1)$	$O(n)$
7	2	$O(1)$
$\boldsymbol{T(n)}$	$13n+22$	$O(n)$

Звідки отримаємо, що її складність має асимптотику $O(n)$.

Як бачимо, для асимптотичної оцінки алгоритму нам фактично не потрібно обчислювати точно час виконання цього алгоритму. Наприклад, якщо певна інструкція знаходиться поза межами циклу, тобто кількість операцій які в ній виконуються не залежить від вхідних даних, то операція виконується за сталий час і асимптотика є $O(1)$. Якщо ж певна операція виконується в циклі, то асимптотика буде напряму залежати від того, скільки разів виконується в циклі операція. Використовуючи такі міркування та скориставшись правилами «великого O», наведеними вище, можна отримати правила для визначення асимптотичної складності алгоритмів.

Правила оцінки алгоритмів

Теорема 2.6. (Послідовна композиція). Найгірший час виконання алгоритму, що складається з послідовності виразів

```
S1
S2
...
Sm
```

має асимптотичну складність

$$O(\max\{T_1(n), \dots, T_m(n)\}),$$

де $T_1(n), \dots, T_m(n)$ – час виконання відповідно інструкцій S1, ..., Sm.

Теорема 2.7. (Цикл while). Найгірший час виконання алгоритму, що містить цикл while

```
while S1:
    S2
```

має асимптотичну складність

$$O(\max\{T_1(n) \times (I(n)+1), T_2(n) \times I(n)\}),$$

де $T_1(n), T_2(n)$ – час виконання відповідно інструкцій S1, S2, а $I(n)$ – кількість ітерацій циклу, що виконується у найгіршому випадку.

Теорема 2.8. (Цикл for). Найгірший час виконання алгоритму, що містить цикл **for**.

```
for i in range(S1):
    S2
```

має асимптотичну складність

$$O(\max\{T_1(n) \times (I(n)+1), T_2(n) \times I(n)\}),$$

де $T_1(n), T_2(n)$ – час виконання відповідно інструкцій S1, S2, а $I(n)$ – кількість ітерацій циклу, що виконується у найгіршому випадку.

Це правило випливає з того, що цикл по колекції еквівалентний такому циклу **while**

```
i = 0
while S1:
    S2
    i += 1
```

Теорема 2.9. (Умовний оператор). Найгірший час виконання алгоритму

```
if S1:
    S2
else:
    S3
```

має асимптотичну складність

$$O(\max\{T_1(n), T_2(n), T_3(n)\}),$$

де $T_1(n), T_2(n), T_3(n)$ – час виконання відповідно інструкцій S1, S2, S3.

Наведемо кілька прикладів застосування вищенаведених правил.

Приклад 2.15. Знайти асимптотичну складність алгоритму знаходження максимального числа у дійному векторі.

Приклад 2.16. Знайти асимптотичну складність алгоритму знаходження максимального числа у матриці.

Приклад 2.17. Знайти асимптотичну складність алгоритму обчислення степеня натурального числа, використовуючи ітеративний та рекурсивний алгоритми.

Реальна ситуація

Асимптотичний аналіз алгоритмів на практиці дозволяє оцінити ефективність вашої програми та дати відповідь на питання чи є сенс застосовувати реалізований алгоритм до ваших вхідних даних. Припустимо, що ви реалізували алгоритм двома способами, перший з яких має складність

$$T_1(n) = O(n^2),$$

а другий

$$T_2(n) = O(n^3).$$

На перший погляд може скластися враження, що це не принциповий момент якій з двох реалізацій віддати перевагу – обидві мають поліноміальний час виконання, обидві однаково швидко працюють для ваших тестових даних (звичайно що тестових даних не дуже великого розміру). Проте реальні показники часу виконання програми для різних розмірів вхідних даних можуть переконати в протилежному – потрібно завжди обирати програму порядок складності якої менший. Нижче наведено таблицю, у якій зазначено реальний час виконання програм, що мають різну асимптотичну складність для вхідних даних різного розміру. Таблиця наведена з розрахунку, що одна елементарна операція виконується за 1нс (одну нано-секунду), і для кожного елементу вхідних даних виконується лише одна елементарна операція.

Складність	$n = 1$	$n = 8$	$n = 1K$	$n = 1024K$
$O(1)$	1 нс	1 нс	1 нс	1 нс
$O(\log n)$	1 нс	3 нс	10 нс	20 нс
$O(n)$	1 нс	8 нс	102 нс	1.05 мс
$O(n \log n)$	1 нс	24 нс	1.02 мс	21 мс
$O(n^2)$	1 нс	64 нс	10.2 мс	18.3 хв
$O(n^3)$	1 нс	512 нс	1.07 с	36.5 р.
$O(2^n)$	1 нс	256 нс	10^{292} р.	10^{10^5} р.

Як бачимо з таблиці, програма, що має складність $O(n^3)$ для вхідних даних розміру 2^{20}, буде виконуватися понад 36 років, у той час як програма зі складністю $O(n^2)$ лише 18 хвилин. Звичайно тут читач може заперечити, апелюючи до того, що можливо його програма не буде оперувати даними таких розмірів і тоді вибір алгоритму не є принциповим. Це дійсно так. Як видно з таблиці, для невеликих розмірів вхідних даних всі вони виконуються за задовільний час. Тут черговий раз важливо наголосити, що коли мова йде про аналіз алгоритму, апріорі передбачається, що алгоритм буде оперувати даними великих розмірів. Більше того, якщо вхідні дані не великого розміру, то розробка складного оптимального алгоритму може себе не виправдати з огляду на витрачений час для його реалізації, відлагодження отриманої програми та подальшу її підтримку. У такому випадку краще обрати алгоритм, що є найпростішим для реалізації.

Завдання для самостійної роботи

2.3. Нехай $f(n) = 3n^2 - n + 4$. Користуючись означенням покажіть що

a) $f(n) = O(n^2)$, b) $f(n) = \Omega(n^2)$.

2.4. Нехай $f(n) = 3n^2 - n + 4$ та $g(n) = n\log n + 5$. Користуючись теоремою 2.1 покажіть що

$$f(n) + g(n) = O(n^2).$$

2.5. Нехай $f(n) = \sqrt{n}$ та $g(n) = \log n$. Користуючись теоремою 2.1 покажіть що

$$f(n) + g(n) = O(\sqrt{n}).$$

2.6. Для кожної пари функцій $f(n)$ та $g(n)$, зазначених у таблиці визначте яке із співвідношень має місце $f(n) = O(g(n))$ чи $g(n) = O(f(n))$

$f(n)$	$g(n)$
$10n$	$n^2 - 10n$
n^3	$n^2 \log n$
$n \log n$	$n + \log n$
$\log n$	$\sqrt[k]{n}$
$\ln n$	$\log n$
$\log(n+1)$	$\log n$
$\log\log n$	$\log n$
2^n	10^n
n^m	m^n
$\cos(n\pi/2)$	$\sin(n\pi/2)$
n^2	$(n\cos n)^2$

2.7. Знайдіть асимптотичний час виконання програми у явному вигляді, якщо для нього відоме рекурентне співвідношення

a) $T(n) = \begin{cases} O(1), & n = 0; \\ aT(n-1) + O(1), & n \geq 1, a > 1. \end{cases}$

b) $T(n) = \begin{cases} O(1), & n = 0; \\ aT(n-1) + O(n), & n \geq 1, a > 1. \end{cases}$

c) $T(n) = \begin{cases} O(1), & n = 0; \\ aT([n/\mathrm{a}]) + O(1), & n \geq 1, a \geq 2. \end{cases}$

d) $T(n) = \begin{cases} O(1), & n = 0; \\ aT([n/\mathrm{a}]) + O(n), & n \geq 1, a \geq 2. \end{cases}$

2.8. Доведіть співвідношення

a) $\sum_{i=0}^{n} i = O(n)$ b) $\sum_{i=0}^{n} i^2 = O(n^2)$ c) $\sum_{i=0}^{n} i^3 = O(n^4)$

d) $\sum_{i=0}^{n} a^i = O(n)$ e) $\prod_{i=1}^{n} \frac{1}{1+i} = O(n)$ f) $\prod_{i=1}^{n} \frac{1}{1+i^2} = O(n^2)$

g) $\prod_{i=1}^{n} \frac{1}{1+i!} = O(n)$ h) $\prod_{i=1}^{n} \frac{1}{1+i^m} = O(nm)$ i) $\prod_{i=1}^{n} \frac{1}{1+i^i} = O(n^2)$

2.9. Чи можна описати алгоритм для кожної з задач підрахунку суми наведених нижче, асимптотична складність яких буде $O(1)$? Якщо так, то наведіть фрагмент такої програми.

a) $\sum_{i=0}^{n} i$ b) $\sum_{i=0}^{n} a^i$ c) $\sum_{i=0}^{\infty} a^i, |a| \leq 1$

2.10. Чи можна описати алгоритм для кожної з задач підрахунку суми наведених нижче, асимптотична складність яких буде $O(n)$? Якщо так, то наведіть фрагмент такої програми.

a) $\sqrt{2 + \sqrt{2 + \cdots + \sqrt{2}}}$ (n коренів); b) $y = x^{2^n} + x^{2^{n-1}} + \ldots + x^4 + x^2 + 1;$

c) $\left(1 + \frac{1}{1^1}\right)\left(1 + \frac{1}{2^2}\right)\ldots\left(1 + \frac{1}{n^n}\right);$ d) $1 + \sin x + \cdots + \sin^n x.$

2.11. Припустимо, що n,m та k невід'ємні цілі числа і методи `e`, `f`, `g` та `h` мають такі характеристики:

- Час виконання у найгіршому випадку методу є `e(n, m, k)` є $O(1)$ і повертає значення з проміжку від 1 до $(n+m+k)$.
- Час виконання у найгіршому випадку методу є `f(n, m, k)` є $O(n+m)$.
- Час виконання у найгіршому випадку методу є `g(n, m, k)` є $O(m+k)$.
- Час виконання у найгіршому випадку методу є `h(n, m, k)` є $O(n+k)$.

Визначте асимптотичні оцінки виконання програм у найгіршому випадку в термінах O–«великого» для фрагментів програм зазначених нижче:

a)
```
1   f(n, 10, 0)
2   g(n, m, k)
3   h(n, m, 1000000)
```

b)
```
1   for i in range(n):
2      f(n, m, k);
```

c)
```
1   for i in range(e(n, 10, 100)):
2      f(n, 10, 0);
```

d)
```
1   for i in range(n):
2      for j in range(i, n):
3         f(n, m, k);
```

2.12. Визначте асимптотичну оцінку виконання функції у найгіршому випадку в термінах O–«великого» для функції:

```
def f(n):
    sum = 0
    for i in range(1, n + 1):
        sum = sum + i
    return sum
```

Що є результатом виконання наведеної функції для заданого натурального числа n? Чи можна оптимізувати цю функцію, покращивши її асимптотичну оцінку?

2.13. Нехай `f(n)` функція визначена у вправі 2.12. Розглянемо функцію

```
def g(n):
    sum = 0
    for i in range(1, n + 1):
        sum = sum + i + f(i)
    return sum
```

Визначте асимптотичну оцінку виконання функції `g(n)` у найгіршому випадку в термінах O–«великого». Що є результатом виконання функції `g(n)` для заданого натурального числа n. Чи можна оптимізувати цю функцію, покращивши її асимптотичну оцінку?

2.14. Нехай `f(n)` функція визначена у вправі 2.12, а функція `g(n)` – у вправі 2.13. Розглянемо функцію

```
def h(n):
    return f(n) + g(n)
```

Визначте асимптотичну оцінку виконання функції `h(n)` у найгіршому випадку в термінах O–«великого». Що є результатом її виконання для заданого натурального числа n. Чи можна оптимізувати цю функцію, покращивши її асимптотичну оцінку?

2.15. Визначте асимптотичну оцінку виконання функції у найгіршому випадку в термінах O–«великого» для функції:

```
def f(n):
    k = 0
    i = n - 1
    while i != 0:
       k += 1.0 / i
       i = i / 2
    return k
```

Що є результатом виконання наведеної функції для заданого натурального числа n?

2.16. Опишіть функції натурального аргументу n, час виконання яких у найгіршому випадку має асимптотику:

a) $O(n)$ b) $O(n^2)$ c) $O(n^3)$
d) $O(\log n)$ e) $O(n \log n)$ f) $O(2^n)$

2.17. Опишіть функції натуральних аргументів n та m, час виконання яких у найгіршому випадку має асимптотику:

a) $O(n+m)$ b) $O(m \log n)$ c) $O(n^m)$

2.18. Натуральне число називається паліндромом, якщо його запис читається однаково зліва направо і справа наліво (наприклад, 1, 393, 4884). Скласти програму, що визначає, чи є задане натуральне число n паліндромом асимптотична складність якої $O(n)$, де n – кількість цифр у числі.

2.19. Числами трибоначчі називається числова послідовність $\{T_k : k \geq 0\}$, задана рекурентним співвідношенням третього порядку:

$$T_0 = 0, T_1 = T_2 = 1, T_k = T_{k-1} + T_{k-2} + T_{k-3}, \qquad k \geq 3.$$

Опишіть функції для обчислення T_n за допомогою рекурентного співвідношення та використовуючи рекурсію. Обчисліть асимптотичну складність кожного з варіантів. Порівняйте абсолютний час виконання (у секундах) обох варіантів для знаходження T_{10}, T_{20}, T_{50}.

2.20. Послідовністю Падована називається числова послідовність $\{P_k : k \geq 0\}$, задана рекурентним співвідношенням третього порядку:

$$P_0 = P_1 = P_2 = 1, P_k = P_{k-1} + P_{k-3}, \qquad k \geq 3.$$

Опишіть функції для обчислення P_n за допомогою рекурентного співвідношення та використовуючи рекурсію. Обчисліть асимптотичну складність кожного з варіантів. Порівняйте абсолютний час виконання (у секундах) обох варіантів для знаходження P_{10}, P_{20}, P_{50}.

РОЗДІЛ 3. ПОШУК ТА СОРТУВАННЯ

§3.1. Пошук

Лінійний (послідовний) пошук

Розглянемо колекцію елементів, що зберігаються у простому списку (масиві). У такому разі ми можемо говоримо, що ці елементи є послідовностями, оскільки кожен елемент зберігається на певній позиції, яка однозначно визначається індексом цього елемента (тобто його номером у послідовності). Оскільки значення індексів впорядковані, то ми маємо можливість послідовно проходити по ним. Цей процес породжує найпростішу, з алгоритмічної точки зору, пошукову техніку – **послідовний** (**лінійний пошук**).

Означення 3.1. **Лінійним** або **послідовним пошуком** називається алгоритм відшукання елемента серед заданого набору шляхом послідовного перебору всіх елементів.

Малюнок нижче демонструє як працює такий пошук. Починаючи з першого елемента в списку, ми послідовно рухаємося по елементах послідовності (у порядку зростання індексів), до тих пір, поки або не знайдемо те, що шукаємо, або не досягнемо останнього елемента. Останнє означає, що послідовність не містить шуканого елемента.

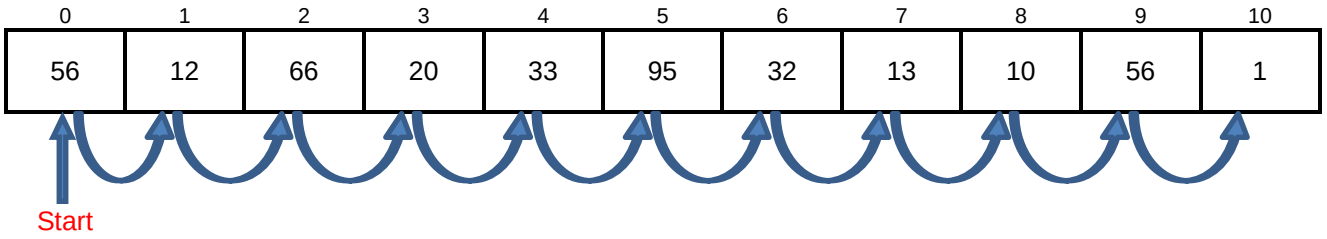


Рисунок 3.1. Лінійний пошук по масиву.

Найпростіша реалізація алгоритму лінійного пошуку, результатом якого буде відповідь чи містить колекція `array` шуканий елемент `x` буде виглядати таким чином

Лістинг 3.1.

```
def linear_search(array, x):
    """ Лінійний пошук у масиві

    :param array: Список елементів
    :param x: Шуканий елемент
    :return: True, якщо шуканий елемент знайдено
    """

    for el in array:
        if el == x:
            return True
    return False
```

Аналіз

Очевидно, що лінійний пошук у найгіршому випадку виконується за лінійний час, тобто $O(n)$. Слід зауважити, що у випадку, якщо шуканий елемент знаходиться на першій позиції, то лінійний пошук здійснюється за сталий час $O(1)$.

Лінійний пошук працює з даними різних типів та не ставить жодних вимог до способу зберігання та розташування даних у колекції – він не вимагає якихось початкових дій по обробці колекції перш ніж розпочати пошук. Проте, він має один суттєвий недолік, який повністю перекреслює його переваги – він занадто повільний, щоб його можна було застосовувати для великих обсягів даних. Дійсно, навряд чи хтось буде користуватися словником у якому слово потрібно шукати послідовно перебираючи всі слова – значно швидше можна знайти потрібне слово користуючись словником у якому слова містяться у лексикографічному порядку. Аналогічно, зайшовши до супермаркету, покупець не буде послідовно перебирати всі товари – він спочатку знайде потрібний відділ, скориставшись картою або

спитавши у консультанта, а вже потім, використовуючи лінійний пошук серед товарів відділу знайде необхідне. У наступних пунктах цього параграфу розглянемо алгоритми, що дозволяють значно пришвидшити процес пошуку даних.

Бінарний пошук

З використанням бінарного пошуку майже напевно зіштовхувалися всі, що хоч раз користувався словником. Оскільки слова у словнику розташовані у лексикографічному порядку, то щоб знайти потрібне слово, словник відкривається посередині і аналізуючи слова, які містяться на відкритій сторінці, приймається рішення у якій з половин словника міститься шукане слово. Далі процедура повторюється спочатку, але для тієї половини словника, що містить слово. Цей процес повторюється доти, доки не буде знайдено потрібне слово або не буде встановлено, що словник його не містить.

Назва такого методу походить від того, що на кожній ітерації, дані, серед яких проводиться пошук діляться навпіл. Бінарний пошук буває двох видів: цілочисельний та дійсний. Цілочисельний пошук застосовується для індексованих масивів даних. Дійсний пошук використовується для пошуку аргументу деякої неперервної функції при якому досягається задане значення.

Цілочисельний бінарний пошук

Припустимо, що ми розглядаємо впорядковану колекцію (список) у якій будь-які два елементи можна порівняти, тобто сказати, який з них більший. Якщо ж елементи у цьому списку, розташовані у порядку зростання або спадання то будемо казати, що список впорядкований (за зростанням або спаданням відповідно).

Бінарний пошук базується та тому, що для будь-якого шуканого числа x і для будь-якого індексу масиву i, ми можемо визначити чи для елементу впорядкованого списку a_i має місце співвідношення

$$a_i \leq x.$$

Для прикладу, розглянемо впорядкований за зростанням список цілих чисел зображений нижче.

0	1	2	3	4	5	6	7	8	9	10
1	10	12	13	20	32	33	56	66	71	95

Припустимо потрібно визначити чи містить цей список (тобто знайти у ньому) число 50. Поділимо список навпіл і визначимо який з двох підсписків може містити шуканий елемент.

0	1	2	3	4	5	6	7	8	9	10
1	10	12	13	20	32	33	56	66	71	95

Для цього досить порівняти центральний елемент 32 (індекс $10/2 = 5$) з шуканим елементом 50.

0	1	2	3	4	5	6	7	8	9	10
1	10	12	13	20	32	33	56	66	71	95

Далі потрібно повторити описану послідовність дій для отриманого підсписку. Ця послідовність дій повторюється доти, доки не лишиться один єдиний елемент:

0	1	2	3	4	5	6	7	8	9	10
1	10	12	13	20	32	33	56	66	71	95

Рисунок 3.2. Бінарний пошук по впорядкованому масиву.

На останньому кроці досить порівняти останній отриманий елемент з шуканим. У нашому прикладі, останній елемент 33 не дорівнює шуканому елементу 50, отже список не місить шуканого числа. Як бачимо, для розв'язання задачі пошуку знадобилося лише три кроки. Проаналізуємо скільки кроків алгоритму у загальному випадку необхідно здійснити для пошуку. Як вже було зазначено, основна ідея бінарного пошуку полягає у тому, що потрібно постійно ділити список навпіл, доки для розгляду не лишиться один останній елемент. Це означає, що алгоритмічна складність алгоритму бінарного

пошуку є $O(\log n)$. Таким чином, якщо список складається з $1K$ елементів, то знайти щось у ньому можна всього за 10 кроків.

Як бачимо, ідея алгоритму надзвичайно проста, тому зразу розглянемо його реалізацію на мові Python. Напишемо програму що лише дає відповідь на питання чи містить відсортований список `array` шуканий елемент `x`. На кожній ітерації будемо запам'ятовувати позицію лівого та правого індексів підсписку, що може містити шуканий елемент.

Лістинг 3.2. Реалізація бінарного пошуку у масиві.

```
def binary_search(array, x):
    """ Бінарний пошук у масиві

    :param array: Список елементів
    :param x: Шуканий елемент
    :return: True, якщо шуканий елемент знайдено
    """
    l = 0                   # Індекс лівого елементу
    r = len(array) - 1      # Індекс правого елементу

    while r > l:

        m = (l + r) // 2    # Індекс середнього елементу
        if x > array[m]:
            l = m + 1
        else:
            r = m

    return array[r] == x
```

Дійсний бінарний пошук

Припустимо задано функцію f на відрізку $[a,b]$, що є неспадною (не зростаючою) на області визначення. Припустимо задано дійсне число c. Задача дійсного бінарного пошуку полягає у тому, щоб знайти найменше таке число $x \in [a,b]$, що $f(x) \geq c$ (або найбільше таке $x \in [a,b]$, що $f(x) \leq c$). Іншими словами, необхідно наближено розв'язати рівняння $f(x) = c$, на проміжку $[a,b]$.

Фактично ідея дійсного бінарного пошуку майже нічим не відрізняється від цілочисельного. Аналогічно до цілочисельного пошуку, будемо на кожній ітерації ділити відрізок $[l,r]$, на якому шукається розв'язок (спочатку $[l,r] = [a,b]$) навпіл і, обчислюючи значення функції у точці поділу $m = (l+r)/2$, визначати у якому з отриманих інтервалів знаходиться розв'язок. Проте, на відміну від цілочисельного бінарного пошуку, постає питання, а коли можна вважати, що ми знайшли розв'язок і що саме вважати розв'язком? Дійсно, як ми знаємо, на будь-якому інтервалі міститься безліч дійсних чисел, а відтак поділ відрізка може відбуватися нескінченно.

Отже, для визначення моменту завершення пошуку використовують три підходи:

1. точність по аргументу;
2. точність по значенню;
3. безпосереднє сусідство двох дійсних чисел.

Розглянемо окремо кожен з них.

Перший підхід пропонує вважати, що пошук завершується у випадку, якщо довжина поточного відрізку $[l,r]$ стає меншою за деяке, наперед визначене додатне число ε

$$r - l < \varepsilon.$$

У цьому випадку розв'язком можна вважати будь-яке значення з останнього інтервалу пошуку – наприклад середину

$$x = \frac{l+r}{2}.$$

Другий підхід пропонує вважати, що пошук завершується у випадку, якщо у середині поточного відрізку $[l,r]$, тобто у точці

$$m = \frac{l+r}{2},$$

функція f має значення, що мало відрізняється від значення c:

$$|f(m) - c| < \varepsilon.$$

При цьому, m вважається розв'язком задачі.

Обидва вищезазначених випадки мають серйозну проблему, яка пов'язана з зображенням дійсних чисел у пам'яті комп'ютера. Може трапитися так, що задана точність ε є такою, що після кількох ітерацій бінарного пошуку, кінці відрізка $[l,r]$ будуть сусідніми дійсними числами. При цьому, очевидно, наступний поділ відрізка навпіл буде породжувати цей же відрізок. Очевидним чином відбудеться зациклення програми.

Читач знайомий з математичним аналізом у цей момент запротестує, апелюючи до того, що з точки зору математики такого не може бути. Дійсно, з математичного аналізу відомо, що дійсні числа щільно заповнюють будь-який відрізок. Проте, на жаль, цього не можливо досягти при моделюванні дійсних чисел у пам'яті комп'ютера – об'єм пам'яті, що виділяється для кожного числа у пам'яті комп'ютера обмежений. Виділеної пам'яті досить для того, щоб змоделювати обмежену множину раціональний чисел, причому цю множину можна впорядкувати за зростанням. Тому, коли мова йде про дійсні числа які обробляються комп'ютером, будемо здебільшого використовувати термін «числа дійсного типу» для того щоб підкреслити, що мова йде про цю множину раціональних чисел, яка моделює дійсні числа. Слід зауважити, що такого зображення дійсних чисел цілком досить для різноманітних математичних задач. Проте з іншого боку, програміст, що має туманне уявлення про зображення чисел у пам'яті комп'ютера приречений на боротьбу з різноманітними помилками, у тому числі із зазначеними вище.

Інша проблема, з якою може зіштовхнутися у процесі реалізації алгоритму бінарного пошуку програміст, це те, що функція f на відрізку $[a,b]$ може дуже швидко зростати (спадати) або дуже повільно зростати (спадати). У першому випадку скоріше за все виникне вищеописана проблема, що кінці відрізка $[l,r]$ на певному етапі перетворяться у два сусідні дійсними числами. У іншому випадку точність знайденого розв'язку буде надзвичайно низькою.

Отже, для того, щоб на практиці подолати описані вище проблеми, пропонуємо використовувати останній третій підхід для визначення моменту завершення пошуку, а саме завершувати пошук, коли кінці відрізка $[l,r]$ будуть сусідніми дійсними числами. Для цього досить перевірити, що середина відрізка $[l,r]$ збігається з одним з кінців l або r.

Реалізуємо алгоритм дійсного бінарного пошуку, що використовує для моменту завершення пошуку останній підхід, а саме безпосереднє сусідство двох дійсних чисел.

Лістинг 3.3. Реалізація бінарного пошуку для знаходження розв'язку рівняння.

```
def binary_continuous(f, c, a, b):
    """ Для монотонної на відрізку [a, b] функції f розв'язує рівняння
                    f(x) = c

    :param f: Монотонна функція - ліва частина рівняння
    :param c: Значення правої частини рівняння.
    :param a: Ліва межа проміжку на якому здійснюється пошук
    :param b: Права межа проміжку на якому здійснюється пошук
    :return: Розв'язок рівняння
    """
    l = a                  # лівий кінець відрізка
    r = b                  # правий кінець відрізка

    m = (l + r) / 2.0      # середина відрізка [l,r]
    while l != m and m != r:
        if f(m) < c:
            l = m  # [l,r] = [x,r]
        else:
            r = m  # [l,r] = [l,x]
        m = (l + r) / 2.0  # середина відрізка [l,r]

    return l
```

Для перевірки роботи алгоритму знайдемо розв'язок рівняння

$$\tan x - 2x = 0$$

на відрізку $\left[\frac{1}{2},\frac{3}{2}\right]$. Легко переконатися, що функція записана у лівій частині рівняння є монотонною, а її значення у на лівому та правому кінцях відрізка мають різний знак. Відтак можемо скористатися вищеописаним алгоритмом бінарного пошуку.

Лістинг 3.3. (продовження).

```
import math
print(binary_continuous(lambda x: math.tan(x) - 2.0 * x, 0, 0.5, 1.5))
```

Результатом виконання вищенаведеного коду буде

```
1.1655611852072112
```

Бінарний пошук по відповіді

ДОПИСАТИ
ДОПИСАТИ
ДОПИСАТИ
ДОПИСАТИ
ДОПИСАТИ

Бінарний пошук є дуже швидким алгоритмом, проте як ми знаємо для його застосування необхідно, щоб колекція була впорядкована, а впорядкування колекції, з алгоритмічної точки зору, це досить не проста задача і як ми побачимо пізніше, найоптимальніші алгоритми сортування мають асимптотичну складність $O(n \log n)$. А тому застосування бінарного пошуку є виправданим (якщо початково колекція невпорядкована) у випадку, якщо необхідно виконати багаторазовий пошук (принаймні двічі), оскільки операції пошуку передує операція сортування колекції.

Хешування та хеш-таблиці

У попередніх пунктах ми розглянули алгоритми пошуку – лінійний та бінарний. Лінійний пошук дуже повільний – він здійснюється за лінійний час. Бінарний пошук є швидким – працює за логарифмічна час – проте, може виконуватися лише для впорядкованих списків.

Спробуємо піти ще на крок далі: побудуємо таку структуру даних, в якій можна буде здійснювати пошук за сталий час $O(1)$. Ця концепція використовує механізм **хешування**.

Хеш-функції

Означення 3.2. **Хешуванням** називається операція перетворення вхідного масиву даних довільної довжини у вихідний бітовий рядок фіксованої довжини за допомогою деякого визначеного алгоритму.

Перетворення вхідного масиву даних відбувається за допомогою спеціальної функції яку називають **хеш-функцією** або **функцією згортки**.

Вхідні дані, на які діє хеш-функція, називаються **ключем** (інколи **повідомленням**). Результат дії хеш-функції на вхідні дані називається **хеш-значенням**, **хеш-кодом** або просто **хешем**. Хеш це натуральне число, яке часто записують у шістнадцятковому вигляді:

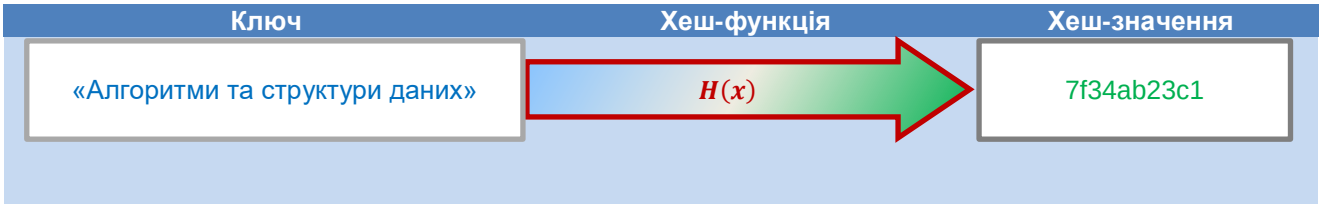


Рисунок 3.3 Хешування рядка.

Приклад 3.1. Прикладом простої хеш-функції, що діє на натуральні числа може бути, функція обчислення остачі від ділення на деяке число, скажімо 11

$$H(x) = x \,\%\, 11$$

Наведемо кілька прикладів обчислення хешів з допомогою цієї функції $H(x)$

Ключ	Хеш (десяткова система числення)	Хеш (шістнадцяткова система числення)

65	10	a
313222	8	8
12	1	1
777	7	7
5625242255631	9	9
1065	9	9

Як бачимо, ця хеш-функція ставить у відповідність натуральному числу, інколи досить великому, ціле число з проміжку від 0 до 10. З наведеного прикладу добре прослідковуються основні найважливіші характеристики яким має задовольняти хеш-функція.

Характеристики хеш-функції

1) На виході хеш-функції завжди отримуються значення фіксованої довжини:

$$H(65) = 10 \leq 10$$
$$H(5625242255631) = 9 \leq 10$$

Навіть, якщо на вхід хеш-функції надійдуть дані велетенської довжини, довжина значення на виході не зміниться. Аналогічна ситуація має місце і для випадку, якщо дані на вході хеш-функції будуть малого розміру. Довжина вихідних значень хеш-функції залежить від типу та явного вигляду хеш-функції. Таке співставлення може бути досить корисним для різних типів задач.

2) Для однакових вхідних даних, хеш-функція незмінно повертає один і той же результат

$$H(65) = 10 = H(65)$$

3) Якщо вхідні дані дуже схожі, проте відрізняються хоча б одним бітом, вихідні дані будуть різними, інколи навіть суттєво різними

$$H(65) = 10$$
$$H(66) = 0$$

4) Трапляється ситуація, коли для цілком різних даних, результатом хеш-функції буде одне значення

$$H(5625242255631) = 9$$
$$H(1065) = 9$$

Така ситуація називається *колізією хеш-функції* і буде детально розглянута нижче.

5) Відновити вхідні дані за їхнім хешем практично неможливо, навіть знаючи явний вигляд самої хеш-функції.
6) Хеш-функція має просту алгоритмічну складність, для того, щоб уникнути зайвого обчислювального навантаження.

Колізії хеш-функції

Якщо уважно подивитися на вищенаведену хеш-функцію можна помітити, що вона може для різних вхідних даних повертати однаковий хеш. Дійсно,

$$H(1) = 1$$
$$H(34) = 1$$
$$H(342) = 1$$

Така ситуація називається колізією хеш-функції.

Означення 3.3. **Колізією** хеш-функції (**хеш-конфліктом**) називається випадок, при якому хеш-функція для різних вхідних блоків даних повертає однакові хеш-коди, тобто

$$x \neq y \Rightarrow H(x) = H(y)$$

Колізії існують для більшості хеш-функцій, що застосовуються на практиці. Проте для «хороших» хеш-функцій частота їхнього виникнення повинна бути близькою до теоретичного мінімуму.

Означення 3.4. **Ідеальною хеш-функцією** називається функція, що відображає кожен ключ з деякого набору ключів у множину цілих чисел без колізій.

Очевидно, що ідеальну хеш-функцію легко побудувати для випадку, коли множина різних вхідних даних є скінченною. Проте на практиці найчастіше виникає ситуація коли потрібно хешувати дані різного розміру за допомогою хеш-кодів сталої довжини. У такому разі не можливо уникнути колізій.

Види хеш-функцій

Підбір хеш-функції залежить від типу даних які мають хешуватися з її допомогою. Наприклад, для цілих чисел часто використовують операцію остача від ділення, а наприклад, для рядків символів – операції що використовують їхні коди.

Можна навести безліч прикладів, проте розрізняють хороші і погані хеш-функції. Хороша хеш-функція повинна задовольняти таким умовам:

- швидко обчислюватися;
- мінімізувати кількість колізій.

Крім цих властивостей часто від хеш-функцій вимагають виконання деяких додаткових вимог, таких наприклад, як неможливість відновлення вхідних даних за їхнім хешем.

Розглянемо які види хеш-функцій застосовуються практиці.

Хеш-функції на основі ділення

Найпростіший спосіб побудови хеш-функції для ключів з множини натуральних чисел полягає у тому, що у ролі хешу натурального числа x використовується залишок від його ділення на M:

$$H(x) = x \,\%\, M,$$

де M — це кількість всіх можливих хешів. На практиці M зазвичай обирають простим. Такий вибір допомагає ефективно розв'язувати проблеми, які виникають під час колізій.

Хеш-функції на базі множення

Інший спосіб побудови хеш-функцій полягає у виборі деякої цілої константи A, що є взаємно простою з w, де w –кількість можливих варіантів значень у вигляді машинного слова (наприклад для 32-бітних операційних системах $w = 2^{32}$). Тоді хеш-функцію можна визначити як

$$H(x) = \left\lfloor M \left\lfloor \frac{A}{w} \cdot x \right\rfloor \right\rfloor.$$

У цьому випадку, як правило M обирають як степінь числа 2.

Хеш-функції на основі методу середніх квадратів

Як і попередні два методи побудови хеш-функції, метод середніх квадратів також застосовується до ключів натурального типу. Цей метод полягає у тому, що значення ключа підноситься до квадрату, а далі з отриманого результату виділяється деяка порція цифр. На базі якої і будується хеш-значення.

Приклад 3.2. Розглянемо ключ 44. Піднесемо його до квадрату і отримаємо $44^2 = 1{,}936$. Виділимо дві середніх цифри отриманого числа – 93. Знайдене число можна вважати хеш-значенням вхідного елементу. Проте, частіше за все до отриманого значення застосовують допоміжну хеш-функцію, наприклад $h_1(x) = x \,\%\, 11$. Тоді, хеш-значення ключа 44 буде дорівнювати 5.

Згортка

Згорткою називається метод побудови хеш-функції, у якому для обчислення хешу вхідний елемент ділиться на складові частини однакового розміру – фрагменти (можливо, крім останнього, який може мати менший розмір). Кожному фрагменту ставиться у відповідність певне ціле значення (як правило за допомогою деякої допоміжної хеш-функції). Отримані значення підсумовуються. Результуюче хеш-значення обчислюється з отриманої суми за допомогою ще однієї допоміжної хеш-функції.

Отже, нехай K – вхідний ключ, $(a_1, \dots a_l)$ – послідовність фрагментів на які розбито ключ. Тоді хеш-функцію можна визначити у такому вигляді

$$H(K) = \big(h_1(a_1) + \cdots + h_l(a_l)\big) \,\%\, M,$$

де $h_i, i = 1, \dots, l$ – послідовність допоміжних хеш-функцій, M – кількість всіх можливих хешів.

Приклад 3.3. Знайдемо методом згортки хеш для ключа, що є телефонним номером

$$K = +380 - 44 - 256 - 0540.$$

Візьмемо послідовність його цифр і поділимо її на групи (фрагменти) по два $(38, 04, 42, 56,\ 05, 40)$. Додаючи отримані фрагменти та обчисливши остачу від ділення на 11 від отриманої суми

$$H(K) = (38 +\ 04 +\ 42 +\ 56 + 05 +\ 40) \,\%\, 11 = 185 \,\%\, 11 = 9$$

отримаємо хеш-значення 9 для вхідного ключа K.

Хеш-функції для рядків

Для побудови хеш-функцій для рядків частіше за все використовується метод згортки. Наприклад, для хешування ключа S, що складається з $l+1$ символів $S = "c_0 c_2 \dots c_l"$, можна запропонувати формулу для обчислення значення хеш-функції у такому вигляді

$$H(S) = \big(h_0(c_0) + h_1(c_1) + \cdots + h_l(c_l)\big) \,\%\, M,$$

де $h_i, i = 0, \dots, l$ – послідовність допоміжних хеш-функцій, що діють на символи рядка, M – кількість всіх можливих хешів.

Приклад 3.4. Розглянемо для прикладу рядок «hash». У ролі усіх допоміжних хеш-функцій $h_i, i = 0, \dots, 3$ будемо використовувати функцію, що повертає код символа:

$$h_i(c_i) = ord(c_i), \qquad i = 0, \dots, 3.$$

Сталу M, як і раніше, виберемо 11. Визначимо коди символів, з яких складається рядок:

символ	h	a	s	h
код	104	97	115	104

Просумуємо отримані значення та знайдемо остачу від ділення на 11 для отриманої суми:

$$H("hash") = (104 + 97 + 115 + 104) \,\%\, 11 = 420 \,\%\, 11 = 2.$$

Таким чином, хеш-значення вхідного рядка «hash» буде 2.

Зауваження. Зазначений у прикладі 3.4 метод побудови хеш-функції для рядків не є хорошим, оскільки різні рядки, що складаються з однакових символів, наприклад «hash» та «hsah» будуть мати однакові хеші, що є незадовільним з точки зору властивостей, яким має задовольняти хеш-функції. Цю проблему можна подолати, якщо послідовність допоміжних функцій визначити так, щоб кожна з допоміжних функцій $h_i, i = 0, \dots, l$ мала явну залежність від позиції символа у рядку на який вона діє. Наприклад, можна модифікувати функції h_i, визначені вище, додавши позицію символа у ролі вагового коефіцієнту

$$h_i(c_i) = (i+1) \cdot ord(c_i), \qquad i = 0, \dots, 3.$$

Інший спосіб побудови хеш-функції для рядків методом згортки, що враховують позицію символів у рядку може бути описаний таким чином. Виберемо деяке просте натуральне число N, що не перевищує 255, наприклад 31. Тоді формула побудови хешу для рядка $S = "c_0 c_2 \dots c_l"$ буде мати вигляд

$$H(S) = \big(ord(c_0) * N^l + ord(c_1) * N^{l-1} + \cdots + ord(c_{l-1}) * N^1 + ord(c_l)\big) \,\%\, M,$$

Таке зображення хеш-функції, при досить великому значенні M дозволяє значно зменшити ймовірність колізій. У вигляді коду ця функція буде мати такий простий вигляд

Лістинг 3.4. Реалізація хеш-функції для рядків.

```
N = 31      # Просте число, що не перевищує 255
M = 100007  # Кількість всіх можливих хешів

def H(S):
    h = 0
    for i in range(len(S)):
        h = h * N + ord(S[i])
    return h % M
```

Крім наведених вище методів хешування рядків, використовують інші методи побудови хеш-функцій, наприклад, алгоритм хешування Пірсона [ ]. З ним та іншими методами, ми пропонуємо познайомитися читачу самостійно.

Хеш таблиці

Одним з застосувань хешування є оптимізація пошуку серед набору даних. Для цього застосовуються хеш-таблиці.

Означення 3.5. **Хеш-таблиця** (eng. hash table, hash-map) це структура даних, що реалізує інтерфейс асоційованого масиву.

Хеш-таблиця дозволяє зберігати пари (ключ, значення) і здійснювати три основні операції:

- додавання нової пари (ключ, значення);
- пошук за ключем;
- видалення за ключем.

Зауваження. Мова програмування Python має вбудовану структуру даних словник (`dict`), яка фактично є хеш-таблицею.

Реалізація хеш-таблиці здійснюється за допомогою масиву наперед визначеного розміру.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

Рисунок 3.4

Як відомо доступ до елементів у масиві здійснюється за допомогою індексу (тобто номеру елемента), який у випадку реалізації хеш-таблиці називають **слотом**. Отже, хеш-таблиця наведена на рисунку вище має 11 слотів, що мають номери від 0 до 10. Кількість слотів хеш-таблиці називається її розміром.

Спочатку хеш-таблиця не містить ніяких елементів, оскільки кожен з них порожній (у Python будемо ініціалізувати їх значенням **None**).

Виконання операцій з хеш-таблицею починається з обчислення значення хешу для ключа. Отримане значення хешу є індексом у масиві, тобто визначає номер слоту де містяться дані або куди треба вставити дані.

Припустимо, що ключі хеш-таблиці є натуральними числами. Тоді, виберемо хеш-функцію

$$H(x) = x \,\%\, 11 \tag{3.1}$$

– остача від ділення ключа на 11. Як ми вже знаємо, число 11 вибрано не випадково – це кількість всіх можливих хешів, що власне і визначає розмір хеш-таблиці. Зауважимо, що як правило, на практиці, операція «остача від ділення» присутня у тій чи іншій формі у всіх хеш-функціях, оскільки хеш-код елементу має обов'язково знаходитися у діапазоні номерів слотів хеш-таблиці.

Припустимо ми хочемо розмістити у хеш-таблиці дані, що мають ключами числа $\{54, 26, 93, 17, 77, 31\}$. Обчислимо їхні хеші

Ключ	Хеш
54	10
26	4
93	5
17	6
77	0
31	9

Нам лишається розмістити вихідні значення за їхніми хеш-кодами у відповідні слоти хеш-таблиці, як показано на рисунку нижче

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Рисунок 3.5

Таким чином, очевидно, що використовуючи хеш-таблиці ми можемо знайти потрібний елемент серед набору за сталий час $O(1)$. Дійсно, для цього нам потрібно лише обчислити хеш для шуканого елементу і звернутися за відповідним індексом до хеш-таблиці. Оскільки кожна з зазначених операцій має складність $O(1)$, то і складність пошуку складає $O(1)$.

При роботі з хеш-таблицями важливу роль відіграє величина, яка називається фактором (коефіцієнтом) заповнення хеш-таблиці. Ця величина визначається як відношення кількості зайнятих слотів (n) до загального числа слотів (N):

$$\lambda = \frac{n}{N}$$

У нашому випадку, після додавання елементів у таблицю, $\lambda = 6/11$. Фактор заповнення хеш-таблиці вважається важливим параметром від якого напряму залежить середній час виконання операцій у хеш-таблиці.

Розв'язання колізій

Зазначена вище техніка коректно працює лише у випадку, якщо кожен елемент претендує на унікальну позицію в хеш-таблиці, тобто різні ключі мають різні хеш-значення. Якщо ж нам, наприклад, потрібно додати у раніше створену хеш-таблицю ключ 44, то виникне колізія. Дійсно, ключ 44 відповідно до нашої хеш-функції має хеш 0. Такий же хеш має ключ 77, який уже міститься у таблиці. Виникає питання: яким чином розмістити у хеш-таблиці новий ключ 44 і чи можливо це взагалі зробити?

Виникнення колізії це досить поширене явище під час процедури хешування. Наприклад, при додаванні у таблицю розміром 365 слотів усього лише 23 елементів, ймовірність колізії вже перевищує 50%[1].

Як ми знаємо, теоретично, уникнути колізій можливо у випадку використання ідеальної хеш-функції. Але, на практиці така ситуація практично неможлива – множина ключів може бути необмеженою, а множина хешів – обмежена, що диктується властивостями хеш-функцій. Таким чином, розв'язання колізій є важливою частиною хешування.

Означення 3.6. Систематичний метод розміщення у хеш-таблиці елементів, що мають однакові хеші називається **процесом розв'язанням колізій**.

Існує два основних типи хеш-таблиць

- з відкритою адресацією,
- з ланцюжками,

які розрізняються способом розв'язання колізій у хеш-таблиці.

Хеш-таблиці з відкритою адресацією

У хеш-таблицях з відкритою адресацією під час розв'язання колізій проводиться пошук іншого вільного місця для розміщення елементу. Щоб це зробити, починаючи з оригінальної позиції (тобто зі слоту, номер якого є хешем елементу), будемо переміщуватися по слотах деяким визначеним способом, доти, доки не буде знайдено порожній слот, власне у який і буде записаний елемент. Зауважимо, що можливо буде необхідним повернутися до першого слоту таблиці циклічно, щоб охопити таблицю повністю. Якщо спосіб пошуку вільного слоту полягає у послідовному відвідуванні всіх слотів, починаючи з оригінального, то такий метод розв'язання колізій називається **лінійним зондуванням**.

Для прикладу, припустимо, необхідно додати по черзі елементи 44, 55, 20 та 13, до раніше створеної хеш-таблиці, заповнення слотів якої зображено на рисунку 3.5. Хеш-значення, обчислене за формулою (3.1) для елементу 44 дорівнює 0. Як бачимо з рисунка 3.5, слот з номером 0 зайнятий іншим елементом. Тоді для елементу 44 шукаємо перший вільний слот – він має номер 1. Розміщуємо туди елемент 44:

0	1	2	3	4	5	6	7	8	9	10
77	44	None	None	26	93	17	None	None	31	54

Рисунок 3.6

У цьому прикладі, фоном будемо виділяти вставлені у таблицю елементи.

Вставимо тепер елемент 55, хеш якого також дорівнює 0. Вільним слотом після оригінального, буде вже слот з номером 2. Після вставки у слот з номером 2 елемента 55, хеш-таблиця набуде вигляду

0	1	2	3	4	5	6	7	8	9	10
77	44	55	None	26	93	17	None	None	31	54

Рисунок 3.7

[1] Відомий факт з теорії ймовірності, що називається «парадокс днів народження».

Для елемента 20, хеш якого дорівнює 9, відповідний слот також зайнятий. Щоб знайти вільний слот методом лінійного зондування, прийдеться почати пошук з початку таблиці, оскільки останній слот таблиці, що має номер 10 також зайнятий. В результаті розміщуємо елемент 20 у слот номер 3.

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Рисунок 3.8

Нарешті, вставка елемента 13 відбудеться аж у позицію 7.

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	13	None	31	54

Рисунок 3.9

Пошук елемента у хеш-таблиці здійснюється аналогічним чином. Припустимо необхідно знайти елемент 77 у таблиці. Його хеш значення буде 0. Звертаючись до слоту з номер нуль виявимо там число 77. Отже шуканий елемент знайдений. Тепер, припустимо, що необхідно знайти у таблиці елемент 55. Його хеш також 0. Проте, слот з номером 0 містить інший елемент. Це не означає, що таблиця не містить елементу 55 – під час заповнення таблиці могла відбутися колізія (що власне і відбулося) і елемент 55 був розміщений у іншому слоті. Починаємо послідовно переглядати всі елементи починаючи з наступної позиції, доки не буде знайдений елемент 55 або не зустрінемо **None**. Останнє буде означати, що таблиця не містить шуканого елементу. Таким чином знаходимо шуканий елементи у позиції 2.

Недоліком лінійного зондування є його схильність до кластеризації – елементи у таблиці групуються. Це означає, що якщо виникає багато колізій з одним хеш-значенням, то слоти, що знаходяться поруч під час лінійного зондування будуть заповнені. Нижче на рисунку, виділено кластер елементів, що мають хеш 0.

0	1	2	3	4	5	6	7	8	9	10
77	**44**	**55**	20	26	93	17	13	None	31	54

Рисунок 3.10

Кластеризація в свою чергу впливає на вставку інших елементів – так ключі 20 та 13 у нашому прикладі були вставлені у позиції, що знаходяться далеко від їхніх хеш-зазначень. Останнє, у свою чергу, значно сповільнює пошук елементів у таблиці, фактично перетворюючи пошук по хеш-таблиці у лінійний пошук.

Щоб подолати проблему кластеризації ключів, описану вище, замість лінійного зондування використовують інші способи пошуку вільних слотів у таблиці – послідовний пошук замінюють таким у якому, частину слотів пропускають (навіть якщо вони були вільні). Наприклад, під час колізії переглядають не кожен наступний слот, а через один. Або використовують не сталий крок пропуску, а такий, що змінюється за деяким відомим законом.

При такому пошуку вільного слоту для розміщення нового значення, відбувається більш рівномірний розподіл елементів, що викликають колізію. Це у свою чергу, зменшує кластеризацію.

Загальна назва для такого процесу пошуку іншого вільного слота після колізії – **повторне хешування**. При цьому використовується функція повторного хешування, яка повертає нове хеш-значення для поточного хешу елементу

$$h_{new} = R(h_{curr}), \tag{3.2}$$

тут h_{curr} – поточний хеш, h_{new} – новий хеш. Наприклад, під час звичайного лінійного зондування, функція повторного хешування матиме вигляд

$$h_{new} = (h_{curr} + 1) \,\%\, l_{table},$$

де l_{table} – розмір хеш-таблиці.

Під час зондування з пропуском через один, функція повторного хешування буде

$$h_{new} = (h_{curr} + 2) \,\%\, l_{table},$$

а якщо величина пропуску s, то функція повторного хешування буде мати вигляд

$$h_{new} = (h_{curr} + s) \% l_{table}.$$

Зауваження. Важливим є те, що величина пропуску s має бути такою, щоб в результаті відвідати усі слоти. Інакше, можливий випадок, що для вставки елементу не знайдеться вільного слоту, притому, що частина таблиці залишиться невикористаною. Найпростішим способом забезпечити виконання цієї умови визначити розмір таблиці простим числом. Саме тому у прикладах наведених вище під час побудови хеш-таблиці використовувалося 11 слотів.

Схема відкритої адресації дозволяє уникнути додаткових витрат пам'яті на розміщення кожного нового елементу, що є її значною перевагою перед іншими способами побудови хеш-таблиць. Проте, недоліком усіх схем відкритої адресації є те, що кількість елементів, які можуть зберігатися у таблиці може досягти кількості слотів у ній. На практиці, навіть з хорошими хеш-функціями, продуктивність серйозно падає, якщо фактор завантаження таблиці наближається до 0.7. Це, в свою чергу, викликає необхідність динамічного збільшення розміру хеш-таблиці з відповідними затратами.

Хеш-таблиці з ланцюжками

Альтернативним методом розв'язання проблеми колізій є така організація хеш-таблиці у якій кожен слот містить не конкретний елемент, а посилання на колекцію елементів (ланцюжок), що мають одне і теж саме хеш-значення.

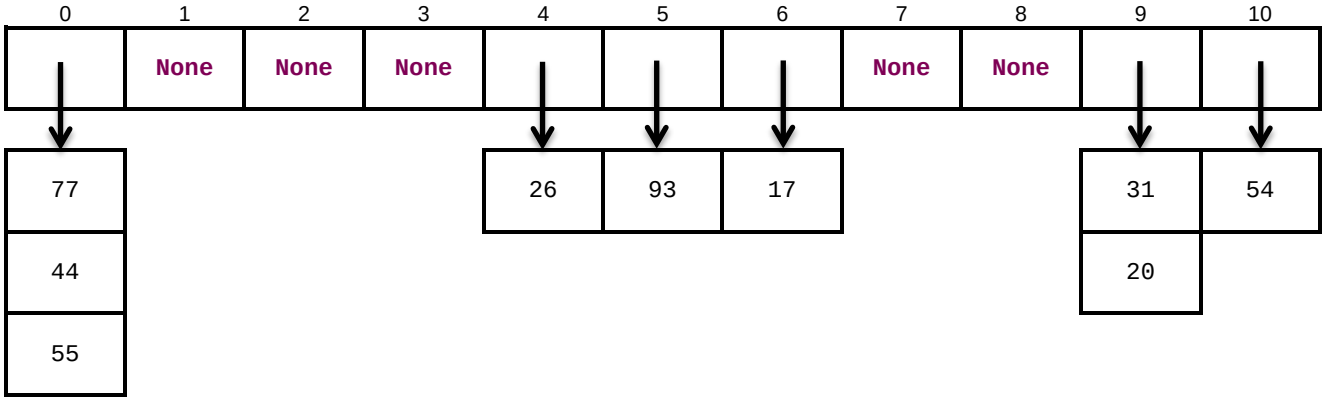


Рисунок 3.11. Хеш-таблиця з ланцюжками.

Хороша реалізація хеш-таблиці вимагає щоб пошук елемента у таблиці та вставка нового елементу до хеш-таблиці відбувалися за сталий час. Тому, як правило, вставка нового елементу у таблицю здійснюється як додавання його в кінець або у початок ланцюжка (залежно від організації структури самого ланцюжка). Наприклад, якщо ланцюжок таблиці реалізовано за допомогою стандартного списку Python, то очевидно, що оптимальним буде додавати нові елементи у кінець відповідного ланцюжка. Якщо ж ланцюжок оформлено у вигляді зв'язного списку, що буде розглянуто пізніше, то вставка здійснюється у його початок.

Нижче на рисунку наведено вставку у таблицю елементів 33, 37 та 57 у кінець відповідного ланцюжка.

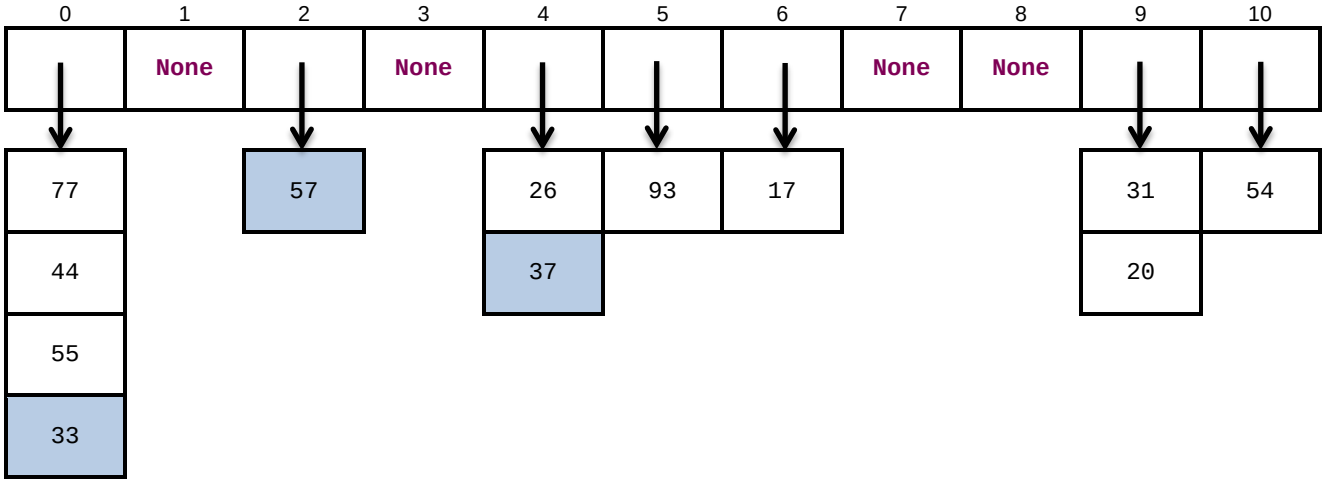


Рисунок 3.12. Додавання елементів до хеш-таблиці

Пошук серед елементів по хеш-таблиці здійснюється лінійним пошуком серед елементів відповідного ланцюжка. Очевидно, що чим більше елементів хешуються в один ланцюжок, тим важче знайти елемент у хеш-таблиці.

Реалізація класу HashTable

Реалізація хеш-таблиці вимагає опису таких основних операції:

- `put(key, value)` – додавання пари (ключ, значення). Якщо такий ключ вже існує, то ця операція заміняє старе значення новим;
- `get(key)` – за заданим ключем повертає значення з таблиці або **`None`**, якщо такий елемент відсутній у таблиці;
- `del` – видаляє пару ключ значення.

Крім цього перевизначимо оператори

- `len()` – кількість пар ключ-значення, що містяться у колекції.
- `in` – оператор перевірки входження ключа у колекцію.

Реалізуємо тип даних `HashTable`. Для розв'язання колізій використаємо алгоритм лінійного зондування.

Лістинг 3.5. Реалізація хеш-таблиці методом лінійного зондування.

```
class HashTable1:
    """Хеш-таблиця у якій колізії розв'язуються методом лінійного зондування."""

    def __init__(self):
        """ Конструктор - ініціалізує таблицю. """
        self.size = 11          # кількість слотів таблиці
        self.current_size = 0  # поточний розмір таблиці
        self.slots = [None] * self.size  # список слотів
        self.data = [None] * self.size   # дані пов'язані з слотом

    def hash(self, key):
        """ Повертає хеш для ключа

        :param key: ключ
        :return: хеш ключа
        """
        return key % self.size

    def rehash(self, prevhash):
        """ Функція повторного хешування, використовується
        у методі лінійного зондування для розв'язання колізій

        :param prevhash: попередній хеш
        :return: новий хеш
        """
        return (prevhash + 1) % self.size

    def put(self, key, value):
        """ Додає пару (ключ, значення) до таблиці

        :param key: ключ
        :param value: значення
        """
        if self.current_size == self.size:
            raise IndexError
        hash = self.hash(key)          # обчислення хешу ключа
        if self.slots[hash] is None:  # якщо відповідний слот вільний
            self.slots[hash] = key    # додаємо ключ
            self.data[hash] = value   # додаємо значення
            self.current_size += 1    # збільшуємо на 1 кількість елементів
        elif self.slots[hash] == key: # слот зайнятий елеметом з ключем key
            self.data[hash] = value   # змінюємо значення, що відповідає ключу
        else:
            # пошук вільного слота у таблиці або слота, що відповідає ключу
            next = self.rehash(hash)
            while self.slots[next] is not None and self.slots[next] != key:
                next = self.rehash(next)

            if self.slots[next] is None: # Знайдено вільний слот
                self.slots[next] = key   # додаємо ключ
                self.data[next] = value  # додаємо значення
                self.current_size += 1   # збільшуємо на 1 кількість елементів
            else:                        # Знайдено слот з ключем key
```

```
                self.data[next] = value  # змінюємо значення, що відповідає ключу

    def get(self, key):
        """ Повертає значення за ключем

        :param key: ключ
        :return: значення
        """
        hash = self.hash(key)           # обчислення хешу ключа
        if self.slots[hash] is None:    # якщо відповідний слот вільний
            return None                 # таблиця не містить ключа
        elif self.slots[hash] == key:   # слот зайнятий елеметом з ключем key
            return self.data[hash]      # повертаємо значення
        else:
            # Пошук слота з ключем key
            next = self.rehash(hash)
            while self.slots[next] is not None and self.slots[next]!=key and next!=hash:
                next = self.rehash(next)

            # Якщо знайдений слот вільний або пройшли таблицю по колу до вихідного слота
            if self.slots[next] is None or next == hash:
                return None       # таблиця не містить ключа

            elif self.slots[next] == key:  # знайдений слот зайнятий елеметом key
                return self.data[next]     # повертаємо значення

    def __setitem__(self, key, value):
        """ Перевизначення оператора [ ] для запису

        :param key: ключ
        :param value: нове значення
        """
        self.put(key, value)

    def __getitem__(self, key):
        """ Перевизначення оператора [ ] для читання

        :param key: ключ
        :return: значення, що відповідає ключу key
        """
        return self.get(key)

    def __len__(self):
        """ Перевизначення вбудованого метода len()

        :return: Кількість елементів у таблиці.
        """

        return self.current_size

    def __contains__(self, key):
        """ Перевизначення оператора in

        :param key: ключ
        :return: True, якщо ключ міситься у таблиці.
        """
        return not (self[key] is None)

    def __str__(self):
        """ Перевизначення вбудованого методу str()

        :return: Зображення таблиці у рядковому вигляді
        """
        return str(self.slots) + '\n' + str(self.data) + '\n'
```

Подальша робота зі створеною таблицею нагадує роботу зі звичайним словником:

Лістинг 3.5.(продовження)

```
M = HashTable1()  # Створюємо таблицю
M.put(55, "zz")   # додаємо пару (56, "zz")
M.put(66, "AA")   # додаємо пару (66, "AA")
M.put(66, "66")   # змінюємо значення за ключем 66
M.put(77, "77")   # додаємо пару (77, "77")

M[56] = "RR"      # M.put(56, "RR")
M[55] = "55"      # M.put(55, "55")

print(M[56])      # print(M.get(56))
print(len(M))
print(62 in M)
```

Завдання для самостійної роботи

3.1. Реалізуйте алгоритм лінійного пошуку всіх елементів списку, що є степенями двійки та визначте асимптотичну складність отриманого алгоритму.

3.2. Припустимо, що список містить елементи, для яких визначена операція $\leq$ (менше або рівно) і елементи у цьому списку розташовані у порядку зростання. Скориставшись цією властивістю списку, реалізуйте алгоритм лінійного пошуку та оцініть його час виконання. Чи буде такий алгоритм оптимальнішим за наведений у цьому параграфі.

3.3. Реалізуйте алгоритм бінарного пошуку з використанням рекурсії. Реалізуйте два варіанти: з використанням оператора зрізу та без нього.

3.4. Знайдіть найменше $x \in [0{,}10]$, що

$$f(x) = x^3 + x + 1 > 5$$

3.5. Змоделюйте на дошці або на окремому листочку роботу з хеш-таблицею, що містить у ролі ключів

a) цілі числа;
b) рядкові літерали;
c) як цілі числа та/або рядкові літерали.

Розгляньте обидва типи хеш-таблиць (з відкритою адресацією та ланцюжками).

3.6. Запропонуйте алгоритм видалення елементів з хеш-таблиці, що використовує метод лінійного зондування для розв'язання колізій.

3.7. Реалізуйте хеш-таблицю, що буде використовувати квадратичне зондування для розв'язання колізій у хеш-таблиці.

3.8. Реалізуйте хеш-таблицю, що буде використовувати метод ланцюжків для розв'язання колізій. Запропонуйте алгоритм видалення елементів з хеш-таблиці, що використовує метод ланцюжків для розв'язання колізій.

3.9. Запропонуйте алгоритм збільшення слотів хеш-таблиці, що використовує лінійне зондування для розв'язання колізій, для випадку, якщо фактор завантаження таблиці перевищує деякий поріг.

3.10. Задача №744 з сайту http://informatics.mccme.ru (informatics.mccme.ru/mod/statements/view3.php?id=601&chapterid=744)

3.11. Задача №745 з сайту http://informatics.mccme.ru (http://informatics.mccme.ru/mod/statements/view3.php?id=601&chapterid=745)

3.12. Використовуючи створену хеш-таблицю реалізуйте програму, що моделює електронний довідник. Реалізуйте пошук відповідного запису за іменем, прізвищем, номером телефону, електронною адресою.

§3.2. Сортування

Задачі сортування мають надзвичайно важливе значення з практичної точки зору. Вони часто є першим кроком підготовки даних для їхнього подальшого аналізу. Наприклад, ми вже знаємо, що алгоритми бінарного пошуку, вимагають впорядкованості даних.

Існує безліч класичних алгоритмів сортування. Деякі з них не надто швидкі, проте мають не великі вимоги по пам'яті, інші навпаки – дуже швидкі проте потребують значних ресурсів пам'яті. У цьому параграфі розглянемо кілька основних класичних алгоритмів сортування, що використовуються для сортування послідовностей чисел.

Бульбашкове сортування

Бульбашкове сортування (також, використовується термін *сортування обміном*, *англ. Bubble sort*) є найпростішим, з алгоритмічної точки зору, алгоритмів сортування. Його ідея полягає у тому, що здійснюється кілька проходів по списку, під час кожного з яких порівнюють пари сусідніх елементів. Якщо елементи стоять не правильно, вони міняються місцями. Кожен прохід по списку ставить наступне найбільше значення на його правильну позицію. Розглянемо кілька проходів бульбашкового сортування для списку елементів наведеного нижче.

56	12	66	20	33	95	32	13	10

Рисунок 3.13. Не відсортований список елементів

Будемо йти зліва на право. Затіняти, клітини, які розглядаються. Більший з двох елементів будемо позначати напівжирним шрифтом.

12	**56**	66	20	33	95	32	13	10
12	56	**66**	20	33	95	32	13	10
12	56	20	**66**	33	95	32	13	10
12	56	20	33	**66**	95	32	13	10
12	56	20	33	66	**95**	32	13	10
12	56	20	33	66	32	**95**	13	10
12	56	20	33	66	32	13	**95**	10
12	56	20	33	66	32	13	10	**95**

Рисунок 3.14. Перший прохід по списку.
Найбільший елемет списку займає потрібну позицію.

Як бачимо у результаті першого проходу по списку, найбільший елемент 95 опинився на останній позиції – це його позиція у відсортованому списку і у подальшому вона змінювати вже не буде. Наступний прохід бульбашкового сортування здійснюється для всіх елементів списку крім елемента 95, що стоїть вже на своїй правильній позиції.

12	20	33	56	32	13	10	**66**	95

Рисунок 3.15. Два найбільших елементи списку займають свої позиціх

В результаті другого проходу по списку, елемент 66 опиниться на останньому місці серед елементів, що розглядалися на цьому проході. Далі процедура проходу здійснюється аналогічно вищенаведеному, для елементів списку без елементів 66 і 95, що вже займають свої правильні позиції. Фактично, в результаті кожного проходу, найбільший елемент, серед елементів по яких здійснюється прохід, опинятиметься (подібно до бульбашки, що підіймається на поверхню рідини) на останній позиції цього підсписку.

Отже, як бачимо, якщо у списку n елементів, то за перший прохід здійснюється $n-1$ операція порівняння. На другій ітерації проводимо вищеописану процедуру для $n-1$ елементів списку (крім останнього, бо він уже знаходиться на своїй позиції). Наступна ітерація проводиться для $n-2$ елементів списку. І так далі, доки не дійдемо до необхідності пройти останні два елементи, що і завершує процедуру сортування.

Реалізація алгоритму

Ідея алгоритму є досить простою, тому наведемо його без додаткових пояснень, крім тих, що виписані як коментарі у вихідному коді алгоритму.

Лістинг 3.6. Реалізація алгоритму сортування «Бульбашкою».

```
def bubble_sort(array):
    """ Реалізує алгоритм сортування "Бульбашкою"

    :param array: Масив (список однотипових елементів)
    :return: None
    """
    n = len(array)
    for pass_num in range(n - 1, 0, -1):
        for i in range(pass_num):
            # Якщо наступний елемент менший за попередній
            if array[i] > array[i + 1]:
                # Міняємо місцями елементи, тобто
                # виштовхуємо більший елемент нагору
                array[i], array[i + 1] = array[i + 1], array[i]
```

Аналіз алгоритму

При аналізі бульбашкового сортування необхідно звернути увагу, що незалежно від початкового порядку елементів, для списку з n елементів буде здійснено $n-1$ прохід. При цьому, на кожному проході буде здійснено $n-j$ операцій порівняння, де j – це номер проходу. Таким чином, очевидно, що складність алгоритму бульбашкового сортування буде $O(n^2)$.

Бульбашкове сортування часто розглядається як найбільш неефективний сортувальний метод, оскільки він повинен переставляти елементи поки не стане відома їх остаточна позиція. Ці "порожні" операції обміну вельми затратні. Однак, оскільки бульбашкове сортування робить прохід по всій несортованій частині списку, воно вміє те, що не можуть більшість сортувальних алгоритмів. Зокрема, якщо під час проходу не було зроблено жодної перестановки, то ми знаємо, що список вже відсортований. Таким чином, алгоритм може бути модифікований, щоб зупинятися раніше, якщо виявляється, що завдання виконане.

Сортування вибором

Алгоритм сортування вибором дуже подібний до бульбашкового сортування, проте є дещо оптимальнішим, оскільки за кожен прохід по списку відбувається лише одна операція перестановки елементів. Його ідея полягає у тому, що на кожному кроці відбувається (лінійний) пошук найбільшого елементу серед невідсортованої частини списку, який переставляється на відповідну позицію (крайню праву/ліву позицію невідсортованої частини списку).

Розглянемо роботу алгоритму на вищенаведеному списку елементів (див Рисунок 3.13). Найбільший елемент, як і раніше, будемо виділяти напівжирним шрифтом, а елементи, що міняються місцями – підсвічуваннями. Отже, в результаті першого проходу по списку, знайдено найбільший елемент 95

56	12	66	20	33	**95**	32	13	10

Рисунок 3.16. Знайдено найбільший елемент списку 95.

який, міняється місцями з останнім елементом у списку 10:

56	12	66	20	33	10	32	13	**95**

Рисунок 3.17. Елементи 10 та 95 міняються місцями.

Як і у випадку бульбашкового сортування, після першого проходу списку, найбільший елемент списку 95 зайняв свою правильну позицію. На другому проході процедура проводиться для елементів списку без елементу 95, результатом чого елементи 66 та 13 поміняються місцями

56	12	13	20	33	10	32	**66**	95

Рисунок 3.18. Елементи 66, 95 зайняли свої позиції.

Отже, як і у випадку бульбашкового сортування, після першого проходу списку, найбільший елемент займає потрібне місце. Після другого проходу – на своє місце стає наступний найбільший елемент. Процес продовжується доки всі елементи не займуть потрібні місця. Для цього потрібно здійснити $n-1$ прохід алгоритму.

Реалізація алгоритму

Лістинг 3.7. Реалізація алгоритму сортування вибором.

```
def selection_sort(array):
    """ Реалізує алгоритм сортування вибором

    :param array: Масив (список однотипових елементів)
    :return: None
    """
    n = len(array)
    for i in range(n - 1, 0, -1):
        # реалізуємо пошук найбільшого елементу
        maxpos = 0
        for j in range(1, i + 1):
            if array[maxpos] < array[j]:
                maxpos = j

        # Міняємо місцями поточний і найбільший елемент
        array[i], array[maxpos] = array[maxpos], array[i]
```

Аналіз алгоритму

Як в алгоритмі бульбашкового сортування, при сортування вибором для списку з n здійснюється $n-1$ прохід алгоритму по списку, на кожному з яких буде здійснено $n-j$ операцій порівняння, де j – це номер проходу. А отже, хоча за кількістю операцій майже напевно, цей алгоритм оптимальніший, за бульбашкове сортування, проте його асимптотична складність така ж, як у бульбашкового сортування – $O(n^2)$.

Сортування вставкою

Сортування вставкою підтримує відсортованою частина елементів. Кожен же наступний елемент вставляється у потрібну позицію, так щоб підсписок лишався відсортованим. Спочатку вважаємо, що підсписок з одного елементу (що знаходиться на нульовій позиції) відсортований. Далі, кожен наступний елемент, на кожному проході вставляється у відповідну позицію. При цьому, може виникнути ситуація, коли для вставки необхідно зсунути частину елементів списку.

Розглянемо ідею алгоритму на прикладі сортування списку зображеного вище на Рисунку 3.13. Починаємо з підсписку, що містить перший елемент списку, який очевидно є відсортованим. Будемо підсвічувати відсортовану частину списку, а елемент, що вставляється напівжирним шрифтом.

56	12	66	20	33	95	32	13	10
12	56	66	20	33	95	32	13	10
12	56	**66**	20	33	95	32	13	10

12	**20**	56	66	33	95	32	13	10

12	20	**33**	56	66	95	32	13	10

12	20	33	56	66	**95**	32	13	10

12	20	**32**	33	56	66	95	13	10

12	**13**	20	32	33	56	66	95	10

10	12	13	20	32	33	56	66	95

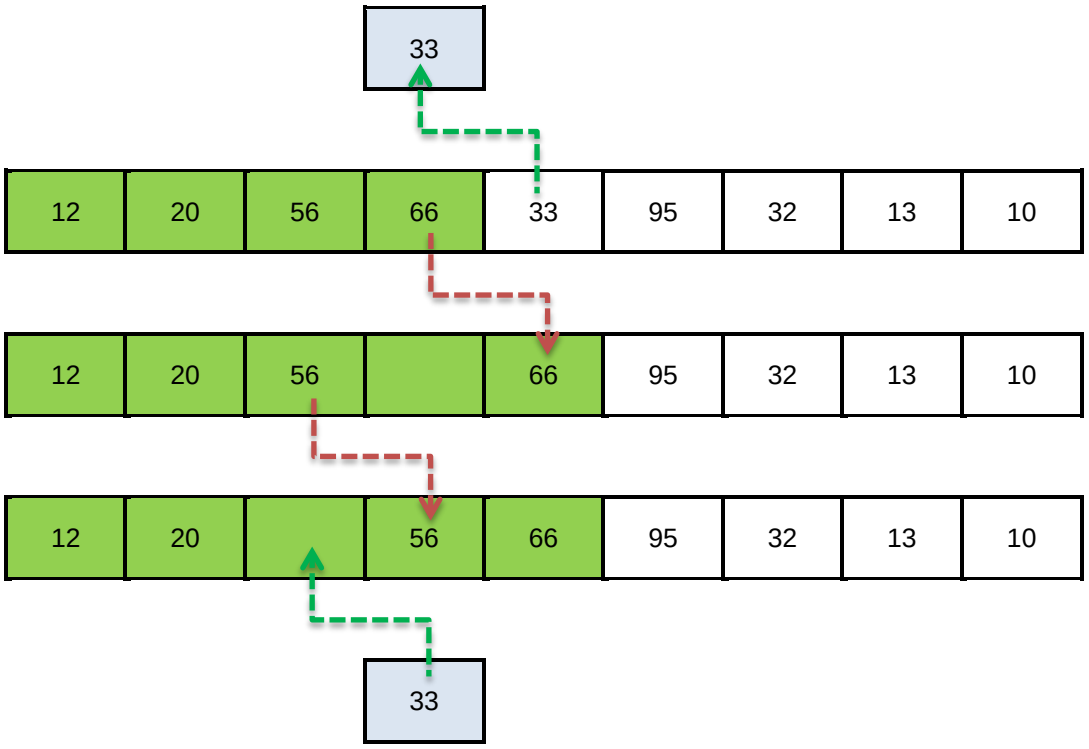
Рисунок 3.19. Сортування вставкою

Реалізація алгоритму

Перш ніж навести повну реалізацію алгоритму, розглянемо його фрагмент, що стосується вставки елементу на потрібну позицію. Наприклад, розглянемо 4-й прохід алгоритму по наведеному вище списку, а саме стан списку наведений нижче, під час вставки елементи 33.

12	20	56	66	**33**	95	32	13	10

Щоб підсписок лишався відсортованим, елемент 33 потрібно вставити на позицію між елементами 20 і 56. Іншими словами, потрібно зсунути вправо елементи 56 і 66, а на місце, що звільнилося вставити елемент 33. Пошук місця вставки і зсув елементів списку будемо здійснювати одночасно. Для цього запам'ятаємо елемент 33. Тоді на його місце (за необхідності) можемо переставити елемент, що розташовується ліворуч.



Отже, напишемо реалізацію алгоритму

Лістинг 3.8. Сортування вставкою.

```
def insertion_sort(array):
    """ Реалізує алгоритм сортування вставкою

    :param array: Масив (список однотипових елементів)
    :return: None
    """
    n = len(array)
    for index in range(1, n):

        currentValue = array[index]  # запам'ятовуємо елемент, що необхідно вставити
        position = index             # та його позицію

        # пошук позиції для вставки поточного елемента
        while position > 0:
            if array[position - 1] > currentValue:
                # зсув елементу масиву вправо
                array[position] = array[position - 1]
            else:
                # знайдено позицію
                break
            position -= 1

        # Вставка поточного елемента у знайдену позицію
        array[position] = currentValue
```

Аналіз алгоритму

Як і у випадках бульбашкового сортування і сортування вибором, алгоритм здійснює $n-1$ прохід для списку з n елементів. Якщо підсписок у який вставляється елемент складається з i елементів, то у найгіршому випадку для вставки нового елементу потрібно i операцій зсуву.

Отже, як і для розглянутих раніше алгоритмів асимптотична складність алгоритму буде $O(n^2)$. Проте, у найкращому випадку, а це буде якщо список вже відсортований, алгоритм буде виконуватися за $O(n)$.

Зауважимо, що взагалі кажучи, операція зсуву вимагає близько третини від обчислювальної роботи обміну, оскільки здійснюється лише одна операція присвоєння. На практиці сортування вставками має дуже хорошу абсолютну швидкодію у порівнянні з бульбашковим сортуванням або сортуванням вибором.

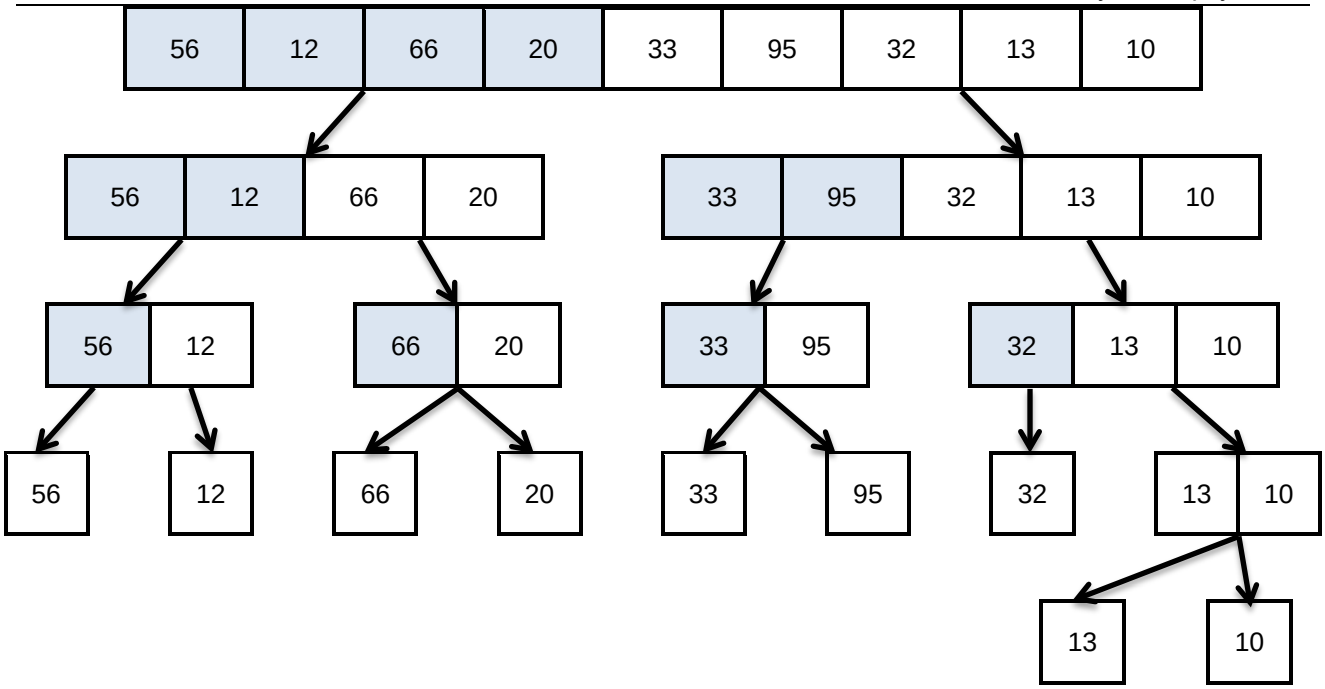
Сортування злиттям

Попередні алгоритми сортування виконувалися у загальному випадку за квадратичний час $O(n^2)$. Розглянемо алгоритми, що базуються на стратегії «розділяй та владарюй», яка дозволяє значно збільшити швидкодію алгоритмів.

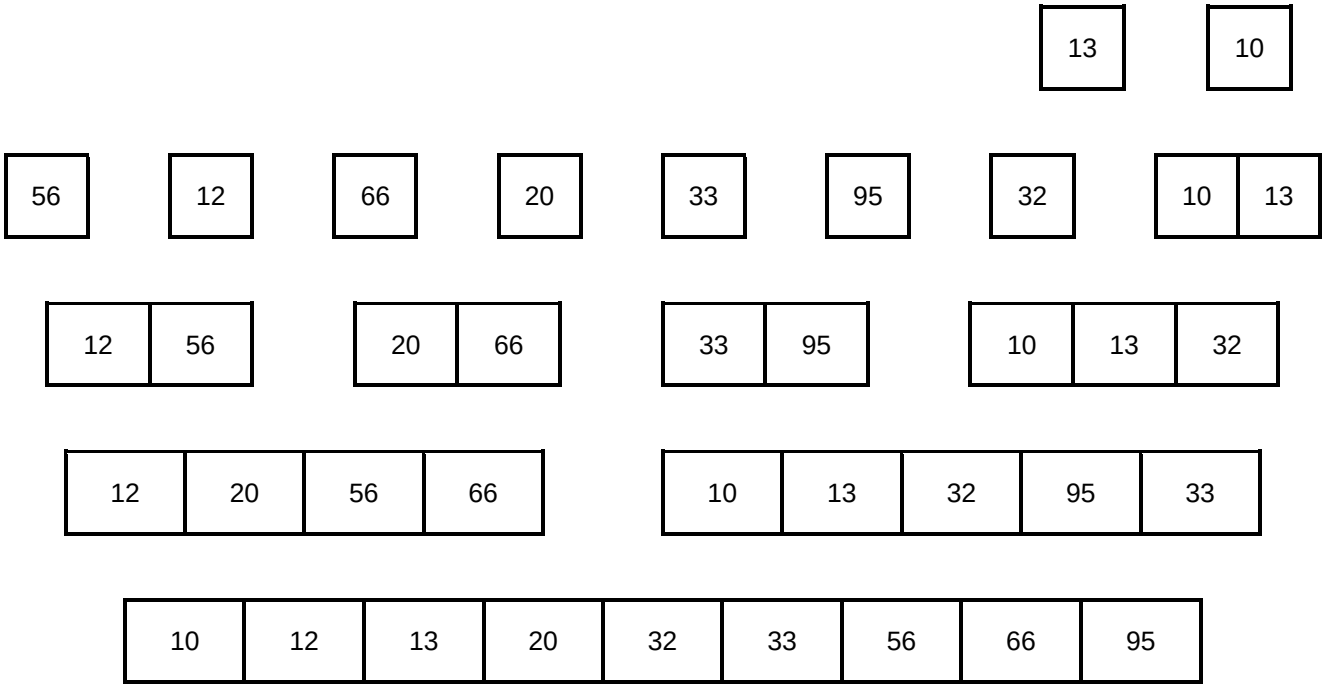
Сортування злиттям – це рекурсивний алгоритм, який можна схематично записати так:

1. Якщо список порожній або складається з одного елементу, то він вважається відсортований.
2. Якщо список складається більше ніж з двох елементів, то він розділяється навпіл, після чого для кожної з половин рекурсивно викликається сортування злиттям. Далі два відсортовані списки об'єднуються (зливаються) у один так, щоб утворений список був відсортованим.

Рисунок нижче демонструє операцію розбиття списку навпіл



Наступний рисунок демонструє процес злиття вже відсортованих списків.



Реалізація алгоритму

Лістинг 3.9. Реалізація алгоритму сортування злиттям

```
def merge_sort(array):
    """ Реалізує алгоритм сортування злиттям

    :param array: Масив (список однотипових елементів)
    :return: None
    """
    print("Splitting ", array)
    if len(array) > 1:
        # Розбиття списку навпіл
        mid = len(array) // 2
        lefthalf = array[:mid]
```

```
        righthalf = array[mid:]

        # Рекурсивний виклик сортування
        # для кожної з половин
        merge_sort(lefthalf)
        merge_sort(righthalf)

        # Злиття двох відсортованих списків
        i = 0
        j = 0
        k = 0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                array[k] = lefthalf[i]
                i += 1
            else:
                array[k] = righthalf[j]
                j += 1
            k += 1

        while i < len(lefthalf):
            array[k] = lefthalf[i]
            i += 1
            k += 1

        while j < len(righthalf):
            array[k] = righthalf[j]
            j += 1
            k += 1
```

Аналіз алгоритму

На кожній ітерації рекурсивного алгоритму ми ділимо список навпіл. Це означає, що для списку з n елементів буде $\log n$ рекурсивних викликів. Крім цього на кожній ітерації буде здійснюватися операція об'єднання двох списків, асимптотична складність якої є $O(n)$. Отже, можемо зробити висновок, що складність алгоритму сортування злиттям є $O(n \log n)$.

Недоліком алгоритму сортування злиттям є потреба у використанні додаткової пам'яті, яка виникає під час ділення списку навпіл, тобто під час створення двох половинок списку, які використовуються для рекурсивного виклику алгоритму сортування.

Швидке сортування

Алгоритм швидкого сортування, використовується для того, щоб отримати ті ж переваги по швидкості, що і сортування злиттям, не використовуючи при цьому додатку пам'ять. Крім цього, однією з особливостей алгоритму є те, що він використовує значно меншу кількість порівнянь і перестановок елементів, ніж інші алгоритми.

Як і алгоритм сортування злиттям, швидке сортування виконується за час $O(n \log n)$. Проте цей час виконання досягається в середньому, а у окремих випадках, алгоритм швидкого сортування може давати навіть гірші результати за бульбашковий алгоритм або алгоритм вставкою.

Розглянемо ідею алгоритму швидкого сортування на прикладі сортування списку зображеного вище на Рисунку 3.13.. На кожній ітерації алгоритму обирається еталонний елемент списку, що називається опорним елементом (який як правило вибирається як крайній лівий або правий елемент списку).

56	12	66	20	33	95	32	13	10

Рисунок 3.20. Обираємо елемент 56 опорним елементом на першій ітерації алгоритму

Далі вихідний список поділяється на дві частини (проте без створення нового списку), одна з яких місить всі елементи які менші за опорний, а інша – елементи що більші. Це досягається з допомогою перестановок елементів, так, щоб зліва від опорного елементу знаходилися елементи списку менші за нього, а справа – більші. Для цього виберемо два маркери – лівий і правий, що спочатку будуть вказувати на крайній лівий та крайній правий елементи списку відповідно (без врахування опорного елементу).

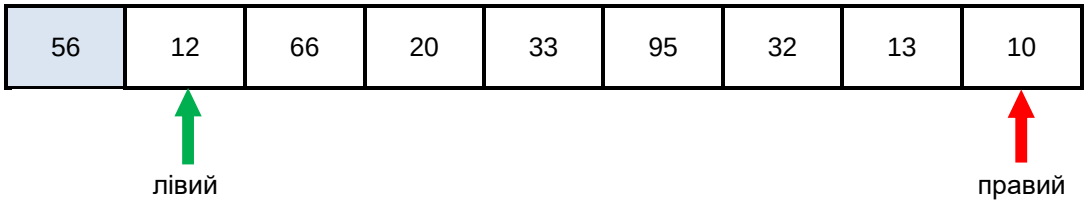


Рисунок 3.21. Визначаємо лівий та правий маркери

Спочатку лівий маркер рухається вправо, доки не зустріне елемент, що є більший або рівний за опорний. У нашому випадку це буде елемент 66.

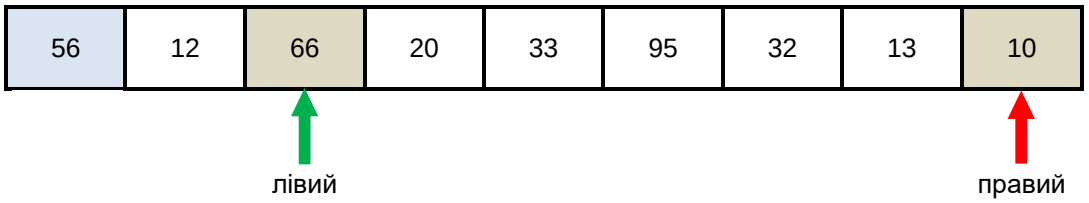


Рисунок 3.22. Лівий досягає вказує на елемент 66.

Після чого правий маркер починає рух вліво доки не зустріне число, що є меншим за опорний елемент. У нашому випадку він з початку вказував на такий елемент. Після того, як обидва маркери зупиняться – елементи на які вони вказують міняються місцями.

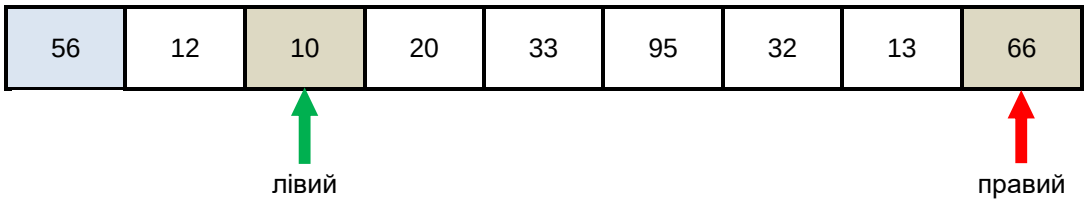


Рисунок 3.23. Лівий досягає вказує на елемент 66.

Далі, лівий маркер починає знову рух вправо, поки не знайде елемент більший за опорний, а правий – вліво поки не знайде елемент менший за опорний.

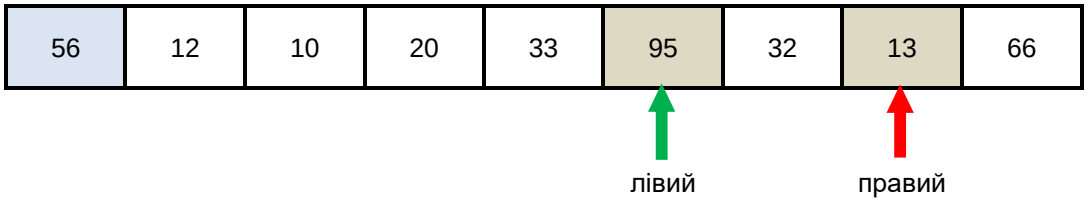


Рисунок 3.24. Лівий маркер рухається вправо, а правий вліво.

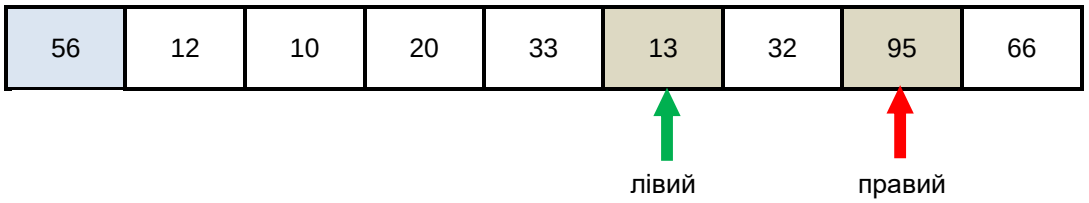


Рисунок 3.25. Міняємо місцями елементи на які вказують маркери.

Рух маркерів, разом з описаною вище процедурою, відбувається доти, доки лівий маркер не займе позицію праворуч від правого.

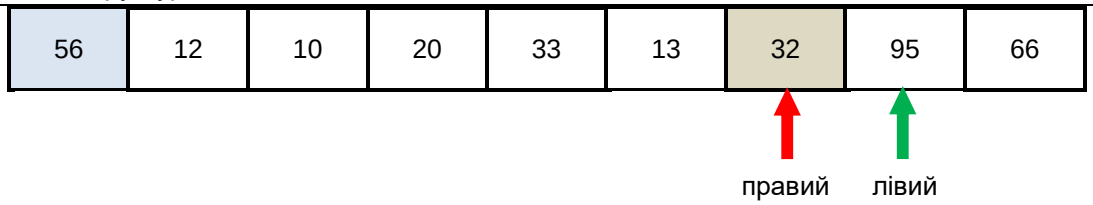
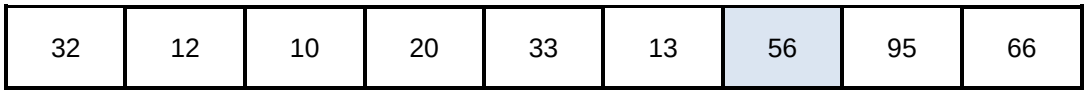


Рисунок 3.26. Лівий маркер займає позіцію правіше правого маркера.

Після цього елемент, на який вказує правий маркер міняється місцями з опорним.



Якщо уважно подивитися на отриманий в результаті такої процедури список, то можна помітити, що зліва від опорного елементу (у лівому підсписку) опинилися всі елементи, що менші за нього, а з права (у правому підсписку) – відповідно більші. Це означає, що опорний елемент зайняв свою позицію і змінювати її більше не буде. Після цього, зазначена операція рекурсивно здійснюється для лівого і правого підсписків.

Реалізація алгоритму

Лістинг 3.10. Швидке сортування

```
def quick_sort(array):
    """ Реалізує алгоритм швидкого сортування

    :param array: Масив (список однотипових елементів)
    :return: None
    """
    quick_sort_helper(array, 0, len(array) - 1)

def quick_sort_helper(array, first, last):
    """ Допоміжний рекурсивний метод,
        що реалізує сортування фрагменту списку обмеженого заданими позиціями

    :param array: Масив (список однотипових елементів)
    :param first: Ліва межа списку
    :param last: Права межа списку
    :return: None
    """
    if first < last:
        # Визначення точки розбиття спику
        splitpoint = partition(array, first, last)
        # Рекурсивний виклик функції швидкого сортування
        # для отриманих частин списку
        quick_sort_helper(array, first, splitpoint - 1)
        quick_sort_helper(array, splitpoint + 1, last)

def partition(array, first, last):
    """ Визначає точку розбиття списку

    :param array: Масив (список однотипових елементів)
    :param first: Ліва межа списку
    :param last: Права межа списку
    :return: Позицію розбиття списку
    """
    pivot = array[first]
    left = first + 1
    right = last
    done = False
    while not done:
        # Рухаємося зліва на право,
        # поки не знайдемо елемент, що більший за опорний
        while left <= right and array[left] <= pivot:
            left += 1

        # Рухаємося справа на ліво,
```

```
        # поки не знайдемо елемент, що менший за опорний
        while array[right] >= pivot and right >= left:
            right -= 1

        # Якщо індекс правого елемента менший за індекс лівого
        if right < left:
            # то розбиття списку завершено
            done = True
        else:
            # міняємо знайдений елементи місцями
            array[left], array[right] = array[right], array[left]

    # ставимо опорний елемент на його позицію
    array[first], array[right] = array[right], array[first]
    return right
```

Аналіз алгоритму

Якщо список складається з n елементів, і якщо точка поділу списку на кожній ітерації рекурсивної функції є приблизно в точкою середини списку, то, очевидно, що складність алгоритму буде близькою до $O(n \log n)$. При цьому алгоритм практично не вимагає додаткової пам'яті. Проте, на жаль, у найгіршому випадку, точка розбиття може бути не посередині, а стрибати зліва на право, роблячи розбиття дуже нерівномірним. У цьому випадку, сортування списку з n елементів розділиться на сортування списків розміром 0 та $n-1$ елемент. Далі 0 та $n-2$ елементи і так далі. Як наслідок, у такому випадку сортування буде здійснюватися за $O(n^2)$. Тому до вибору точки розбиття потрібно підходити системно, аналізуючи дані, що містяться у масиві. Наприклад, можна провести попередній аналіз масиву, що матиме лінійну складність, та знайти його медіану (середній елемент масиву – не плутати з середнім арифметичним). Якщо обирати такий елемент у якості опорного, на кожному виклиці рекурсивного методу, то це гарантуватиме, що складність алгоритму буде близькою до $O(n \log n)$.

Завдання для самостійної роботи

3.13. Модифікуйте алгоритм бульбашкового сортування згідно з наведеною вище пропозицією оптимізації.

3.14. Створіть список з 500 цілих, використовуючи генератор випадкових чисел. Проведіть порівняльний аналіз, використовуючи алгоритми сортування цієї теми. Які відмінності в абсолютній швидкості виконання?

3.15. Бульбашкове сортування може бути змінене, щоб "пузиритися" в обох напрямках. Перший прохід переміщує "вгору" списку, другий - "донизу". Цей процес завершується щойно вичерпається необхідність в проходах.

3.16. Реалізуйте функцію mergeSort без оператора зрізу.

3.17. Реалізуйте алгоритми сортування комплексних чисел за зростанням радіус-вектора.

3.18. Реалізуйте алгоритми сортування об'єктів використовуючи деякий заданий критерій для їхнього порівняння.

3.19. Визначити процедуру впорядкування рядків дійсної матриці за:

а) неспаданням їхніх перших елементів;

б) незростанням сум їхніх елементів;

в) неспаданням їхніх найменших елементів;

г) незростанням їхніх найбільших елементів.

Використати методи обмінного сортування та сортування злиттям.

3.20. Дано масив з n точок площини, заданих своїми координатами. Упорядкувати точки за неспаданням відстані від початку координат.

3.21. Знайдіть у літературі інші, крім розглянутих у цій темі алгоритми сортування. Проаналізуйте їхні переваги на недоліки в порівнянні з іншими алгоритмами сортування.

РОЗДІЛ 4. ПОВНИЙ ПЕРЕБІР

§4.1. Повний перебір

При розв'язанні алгоритмічних задач, найпростішим з концептуальної точки зору (проте не з точки зору реалізації), є метод повного перебору, коли серед множини всіх можливих розв'язків потрібно обрати правильний. Проте, коли мова заходить про оптимальність такого розв'язання, частіше за все побутує думка, що повний перебір є найгіршим з можливих підходів розв'язання. Так, частково це правда – найчастіше повний перебір застосовують у тих випадках, коли або іншого варіанту не лишається і множина всіх можливих розв'язків є не великою, або можна заздалегідь відкинути значну частину хибних розв'язків. Проте, існує широкий спектр алгоритмічних задач, метод повного перебору для яких є досить оптимальним. Раніше ми познайомилися з алгоритмом сортування злиттям, котрий концептуально є нічим іншим, як методом повного перебору і при цьому він є чи не найоптимальнішим з усіх наведених у цьому посібнику алгоритмів сортування масивів. Отже, складність повного перебору залежить від кількості всіх можливих розв'язків задачі, від порядку розгляду цих розв'язків, а також чи можливо відкинути значну частину заздалегідь неправильних варіантів.

Отже,

Означення 4.1. **Повний перебір** (англ. brute force, метод «грубої сили») – загальний метод розв'язання математичних задач шляхом перебору усіх можливих потенційних варіантів.

Фактично, повним перебором можна скористатися лише тоді, коли ми маємо справу з дискретною системою, стани якої можуть бути легко проаналізовані.

Загальних методів побудови алгоритмів, що використовують механізм повного перебору, взагалі кажучи, не існує – у кожному конкретному випадку може бути застосований той чи інший підхід. Проте, очевидно, всі вони мають спільну рису – потрібно організувати певним чином послідовний перебір всіх можливих станів системи, не пропустивши жодного. Часто алгоритм такого перебору станів системи виникає сам собою після детального аналізу дискретної системи, що розглядається у задачі. Наприклад, якщо потрібно побудувати всі можливі комбінації чисел, що складаються з цифр 0 та 1, довжина яких не перевищує задану кількість цифр, то, згадавши про двійкове зображення натуральних чисел, можна застосувати такий алгоритм

Лістинг 4.1. Пошук n-значних чисел, що складаються з 0 та 1.

```
n = int(input())  # кількість цифр

l = 1 << n - 1    # найменше n-значне число з нулів та одиниць
r = 1 << n        # найменше (n+1)-значне чило з нулів та одиниць

# послідовність чисел з проміжку [l, r) у двійковій системі числення буде
# будувати всі можливі комбінації n-значних чисел, що складаються з 0 та 1.
for i in range(l, r):
    print(bin(i))
```

Результатом роботи наведеної програми для числа введеного числа 3 буде

```
3
0b100
0b101
0b110
0b111
```

У цьому прикладі, для повного перебору досить було застосувати послідовний перебір натуральних чисел з заданого проміжку. Якщо ж задача полягає у тому, що необхідно побудувати всі можливі комбінації деякої послідовності з повторами елементів, при цьому кількість елементів наперед відома і фіксована, то найпростішим способом організації такого перебору буде використання вкладених циклів. Для прикладу напишемо програму, що визначає кількості тризначних натуральних чисел, сума цифр яких дорівнює n ($n \geq 1$). Звичайно, можна як і у попередньому прикладі пробігтися одним циклом по всіх натуральних числах від 100 до 999, розбиваючи на кожному кроці число на

сотні, десятки і одиниці. Проте, таке розбиття буде використовувати додаткові операції цілочисельного ділення та остачі від ділення. Тому підемо іншим шляхом, а саме організуємо перебір використовуючи три вкладених цикли – для перебору сотень, десятків та одиниць.

Лістинг 4.2. Пошук всіх тризначних чисел, сума цифр яких дорівнює заданому числу.

```
n = int(input())
counter = 0                     # лічильник кількості чисел
for i in range(1, 10):          # Перебір сотень (від 1, бо інакше число не тризначне)
    for j in range(10):         # Перебір десятків
        for k in range(10):     # Перебір одиниць
            if i + j + k == n:
                counter += 1

print(counter)
```

Хоча цикли часто дають чи не найкращий підхід до повного перебору для широкого класу задач, проте загалом на практиці, найчастіше повний перебір реалізовують використовуючи рекурсивний опис функції. Однією з задач, що розв'язується методом повного перебору є задача відшукання всіх можливих перестановок елементів заданої послідовності. Розв'яжемо її у такій інтерпретації:

Приклад 4.1 [e-olimp, 2169]. За заданим натуральним числом n виведемо усі перестановки з цілих чисел від 1 до n у лексикографічному порядку.

Для того, щоб краще зрозуміти запропонований нижче алгоритм, випишемо всі можливі комбінації для $n = 1,2,3$. Розглянемо найпростіший випадок $n = 1$. Тобто нам потрібно побудувати всі можливі перестановки елементів послідовності $[1]$. Очевидно, існує лише один варіант такої перестановки – це власне цей один елемент і утворює цю перестановку.

Тепер розглянемо $n = 2$. Як ми знаємо з комбінаторики, таких комбінацій є $2! = 2$. Скористаємося цим фактом, щоб переконатися, що виписано всі комбінації. Очевидно, що всі такі комбінації записуються таким чином

$$1\ 2$$
$$2\ 1$$

тобто дописуванням елементу 2 зліва та справа від числа 1.

Далі, випишемо всі можливі комбінації для числа 3, тобто для набору $[1, 2, 3]$.. Кількість різних комбінацій вже буде $3! = 6$.

$$1\ 2\ 3$$
$$1\ 3\ 2$$
$$2\ 1\ 3$$
$$2\ 3\ 1$$
$$3\ 1\ 2$$
$$3\ 2\ 1$$

Очевидно, що ці послідовності було отримано вставкою цифри 3 у всі можливі позиції для послідовностей записаних для попереднього випадку.

Досить часто, при продумуванні алгоритму повного перебору, виписані варіанти для найпростіших випадків наштовхують на ідею алгоритму. Так, якщо уважно придивитися, то, використовуючи отриманий результат, можна продовжити для $n = 4$, вставкою цифри у всі можливі позиції щойно отриманого набору послідовностей. Отже

Лістинг 4.3. Пошук всіх перестановок заданої послідовності

```
def sequences(lst : list, k, n):
    """
    :param lst: підсписок перестановок
    :param k:   елемент для вставки
    :param n:   найбільший елемент послідовності
    """
    if k > n:  # Якщо всі елементи вже вичерпано
        print(*lst)
        return

    # Вставляємо елемент k у всі можливі позиції списку
    # отриманого на попередніх ітераціях
    for pos in range(k):
        lst_next = lst[:]              # Копіюємо список
```

```
        lst_next.insert(pos, k)       # вставляємо елемент
        sequences(lst_next, k + 1, n) # Запускаємо рекурсивно додавання наступного числа.

# Головна програма
if __name__ == "__main__":
    n = int(input())
    lst = []
    sequences(lst, 1, n)
```

Результатом наведеної програми для введеного числа 3 буде

```
3 2 1
2 3 1
2 1 3
3 1 2
1 3 2
1 2 3
```

Як бачимо програма дійсно побудувала всі варіанти, проте поставлена задача не виконана – потрібно вивести послідовності у лексикографічному порядку. Звичайно ніщо нам не заважає відсортувати отриманий список послідовностей, Проте, щоб уникнути зайвих витрат процесорного часу, перепишемо програму, застосувавши інший підхід. А саме: будемо не вставляти новий елемент на всі можливі позиції, а додавати елемент в кінець списку, якщо він ще не міститься у ньому.

Лістинг 4.4. Пошук всіх перестановок заданої послідовності. Видення у лексикографічному порядку.

```
def sequences(lst : list, n):
    """
    :param lst: підсписок перестановок
    :param n:   найбільший елемент послідовності
    :return:    None
    """
    k = len(lst)  # Визначаємо кількість членів поточної підпослідовності

    if k == n:    # Якщо всі елементи вже вичерпано
        print(*lst)
        return

    for i in range(1, n+1):
        if i not in lst:            # Якщо елемент i не міститься у підпослідовності
            lst_next = lst[:]       # Копіюємо список
            lst_next.append(i)      # Додаємо до нього елемент i
            sequences(lst_next, n)  # Додавання рекурсивно інших членів послідовності.

# Головна програма
if __name__ == "__main__":
    n = int(input())
    lst = []
    sequences(lst, n)
```

Однією з класичних задач, що розв'язуються повним перебором, є задача про рюкзак. Розв'яжемо її в такій постановці.

Приклад 4.2 [e-olimp, 2103]. Вася зібрався у похід з друзями-програмістами і вирішив відповідально підійти до вибору того, що він візьме з собою. У Васі є n речей, які він міг би взяти з собою у рюкзаку. Кожна річ важить 1 кілограм. Речі мають різну "користність" для Васі. Похід очікується досить тривалий, і Вася хотів би носити рюкзак вагою не більше w кілограм. Допоможіть йому визначить максимальну сумарну "корисність" предметів у нього в рюкзаку при вазі рюкзака не більше w кілограм.

Оскільки системи станів задачі є дискретною, її можна розв'язавши, повністю перебравши всі можливі розв'язки. Отже, у нас є n речей, які можна укладати в рюкзак. Для кожного предмета існує 2 варіанти: предмет або кладеться в рюкзак, або ні. Тоді, як ми знаємо, алгоритм, що використовує повний перебір всіх можливих варіантів має складність $O\,(2^n)$. Це дозволяє його використовувати лише для невеликої кількості предметів. З ростом кількості предметів задача стає нерозв'язною даним методом за прийнятний час.

Тоді, рекурсивна функція побудови всіх варіантів та підрахунку поточної вартості рюкзака матиме вигляд

Лістинг 4.5.Задача про рюкзак.

```
def max_score(weight, score, num):
    """
    :param weight: поточна вага рюкзака
    :param score:  поточна вартість рюкзака
    :param num:    номер предмета
    """
    global maxScore, W, n    # глобальні змінні:
                             # maxScore - поточна максимальна вага рюкзака,
                             # W - максимальна вага рюкзака, n - кількість предметів

    # досягли максимальної глибини рекурсії
    if weight == W or num >= n:    # якщо вага рюкзака W або речі закінчилися
        if score > maxScore:       # порівнюємо поточну вартість рюкзака
                                   # з поточною максимальною вартістю знайденою раніше
            maxScore = score       # зберігаємо оптимальний розв'язок (при необхідності)
        return

    # будуємо наступні варіанти
    max_score(weight,     score,          num + 1)  # предмет не кладемо у рюкзак
    max_score(weight + 1, score + a[num], num + 1)  # предмет кладемо у рюкзак
```

Виклик цієї функції буде мати вигляд

Лістинг 4.5. (продовження)

```
maxScore = 0                           # максимальна вартість рюкзака
W, n = map(int, input().split())       # вага рюкзака і кількість різних речей
a = list(map(int, input().split()))    # список цінностей речей
max_score(0, 0, 0)                     # старт рекурсивної функції
print(maxScore)                        # виведення результату
```

§4.2. Метод гілок та меж

Чи не найбільшою групою задач, які розв'язуються методом повного перебору є задачі дискретної оптимізації. Такі задачі мають скінченну множину допустимих розв'язків, які теоретично можна перебрати і вибрати найкращий, тобто той, що дає мінімум (або максимум) цільової функції. На практиці ж такий перебір може стати нездійсненним навіть для задач, дискретна система станів якої є невеликою. Тому постає задача оптимізації підходу повного перебору. Така оптимізація, завжди використовує властивості задачі, що розв'язується і полягає у тому, щоб так організувати перебір, щоб відкинути значну частину хибних розв'язків. Такі методи перебору називаються методами неявного перебору і, найбільш поширений з них – метод гілок і меж.

Метод гілок і меж базується на двох процедурах – розгалуженні та знаходженні оцінок (меж). Процедура розгалуження полягає у тому, що множина допустимих розв'язків розбивається на підмножини меншого розміру. На кожному кроці елементи такого розбиття аналізуються на предмет того, чи містить дана підмножина оптимальний розв'язок чи ні. Якщо розглядається задача знаходження мінімуму цільової функції, то така перевірка здійснюється шляхом обчислення оцінки знизу для цільової функції на даній підмножині. Якщо оцінка знизу не менша за рекорд (найкращий, зі знайдених розв'язків), то підмножина може бути відкинута, оскільки вона не містить розв'язку, що кращий за рекорд. Якщо значення цільової функції на знайденому розв'язку є меншим за рекорд, то відбувається зміна рекорду. Якщо всі елементи розбиття вдається відкинути, алгоритм завершає свою роботу, а поточний рекорд є оптимальним розв'язком. Інакше, серед підмножин, що залишилися обирається найперспективніша, наприклад з найменшим значенням нижньої оцінки, для якої проводиться поділ на підмножини. Нові підмножини знову аналізуються. Ця процедура проводиться (рекурсивно) доти, доки не буде знайдено оптимальний розв'язок.

Задача знаходження максимуму цільової майже повністю повторює вищенаведений опис, за виключенням того, що аналізується оцінка зверху.

Для демонстрації алгоритму розглянемо задачу

Приклад 4.3 [e-olimp, 3533]. Василько просто у захваті від гри "Вормікс". Він досягнув вже значного рівня, тож може відкривати бій із босом. Щоб відкрити новий бій, йому потрібно набрати не менше, ніж K очок за місії. Відомо, що всього є N місій. Для кожної місії відомо скільки часу триватиме її проходження і скільки за неї буде нараховано очок. Також відомо, що Василько є дуже добрим гравцем, а отже він з легкістю зможе пройти будь-яку місію. На жаль, він немає часу, щоб пройти всі

місії, але дуже хоче відкрити бій з босом, тож він хоче дізнатися за який мінімальний проміжок часу він зможе набрати не менше K очок.

Як і у задачі про рюкзак, система станів задачі є дискретною, а отже її можна розв'язати повністю перебравши всі можливі розв'язки. Як і у випадку задачі про рюкзак, у цій задачі кожна місія може бути врахована або не врахована. Проте, оптимізуємо цю задачу скориставшись методом гілок і границь. А саме, будемо враховувати кожну наступну місію, лише у тому разі, якщо сумарний час усіх попередніх місій разом з поточною не перевищує мінімального значення часу на деякому розв'язку (рекорду).

Отже, рекурсивна функція знаходження мінімального часу проходження місій буде мати вигляд

Лістинг 4.6. Метод гілок та меж.

```
minTime = 100500   # ініціалізація значення рекорду

def findMinTime(time, score, mission_num):
    """
    :param time:        поточне значення часу
    :param score:       поточний рахунок
    :param mission_num: номер місії
    """
    global minTime  # глобальна змінна, що містить рекорд

    # Термінальна гілка, якщо опрацьовані всі місії
    if mission_num >= N:
        # Якщо значення цільової функції на знайденому розв'язку є меншим за рекорд
        if score >= K and minTime > time:
            minTime = time  # зміна рекорду
        return

    # рекурсивний виклик без урахуванням місії mission_num
    findMinTime(time, score, mission_num + 1)

    nextTime = time + t[mission_num]
    # Якщо оцінка знизу не менша за рекорд, то підмножина може бути відкинута
    if nextTime >= minTime:
        return

    nextScore = score + a[mission_num]  # рахунок з урахування місії mission_num
    # рекурсивний виклик з урахуванням місії mission_num
    findMinTime(nextTime, nextScore, mission_num + 1)
```

Зчитування даних задачі та виклик цієї функції буде мати вигляд

Лістинг 4.6. (продовження)

```
maxN = 100
a = [0] * maxN
t = [0] * maxN

# зчитування даних задачі
N, K = map(int, input().split())
# зчитування вартостей місій та часу їхнього проходження
for i in range(N):
    a[i], t[i] = map(int, input().split())

findMinTime(0, 0, 0)      # старт рекурсивної функції

# Виведення результату
if minTime == 100500:
    print(-1)
else:
    print(minTime)
```

§4.3. Метод «Розділяй і володарюй»

Одним з найважливіших методів, що широко застосовується при проектуванні ефективних алгоритмів є метод, що називається **методом декомпозиції** (або на жаргоні алгоритмістів – метод

«розділяй і володарюй»). Цей метод передбачає поділ вихідної задачі на дрібніші задачі, розв'язання яких з алгоритмічної точки зору має меншу складність, на основі розв'язків яких можна легко отримати розв'язок вихідної задачі. При цьому, структура кожної з підзадач є подібною до структури вихідної, що в свою чергу дозволяє далі поділили їх на підзадачі аж доки не дійдемо до підзадач розв'язання яких є тривіальним.

Отже, будь-який алгоритм, що розв'язує задачу методом декомпозиції складається з трьох кроків:

1. Розбиття задачі на кілька простіших незалежних між собою підзадач;
2. Розв'язання кожної з підзадач (явно у тривіальних випадках або рекурсивно);
3. Об'єднання отриманих розв'язків підзадач.

Як бачимо реалізація концепції «розділяй та володарюй» має рекурсивний характер. Враховуючи специфіку методу, оцінка часу виконання завжди буде зображуватися у вигляді рекурентної формули

$$T(n) = aT(n/b\ ) + f(n)$$

де, a – кількість підзадач, n/b – розмір підзадач, $f(n)$ – час, що витрачається на декомпозицію задачі та об'єднання результатів розв'язків підзадач.

Прикладами застосування цього методу є вивчені раніше алгоритми сортування злиттям та бінарний пошук. Іншим прикладом застосування цього методу є підрахунок елементів послідовності чисел Фібоначчі використовуючи рекурентні формули або рекурсивний опис.

Метод «розділяй і володарюй», фактично, є різновидом концепції повного перебору.

Завдання для самостійної роботи

4.1.

РОЗДІЛ 5. ЛІНІЙНІ СТРУКТУРИ ДАНИХ

Ми вже знайомі з деякими структурами даних, такими як списки, словники, множини, які входять до бази Python. Розглянемо інші, більш складні структури даних, без розуміння яких не можна уявити сучасне програмування.

Спочатку розглянемо п'ять простих, проте дуже потужних концепцій – стек, черга, дек, пріоритетна черга та зв'язний список. Ці структури є прикладами колекцій, елементи яких впорядковані в залежності від того, як вони додавалися або віддалялися. Доданий, елемент залишається на одному і тому ж місці по відношенню до решти, що були додані у колекцію раніше або пізніше нього. Колекції такого роду часто називають **лінійними структурами даних**.

§5.1. Стек

Означення 5.1. **Стек** – це динамічна структура даних із принципом до-ступу до елементів "останнім прийшов – першим пішов" (англ. LIFO – last in, first out). Усі операції у стеку проводяться тільки з одним елементом, що називається **верхівкою стека**, та був доданий до стеку останнім.

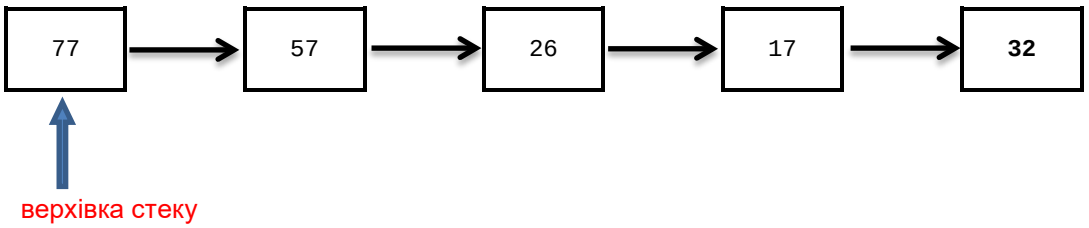


Рисунок 5.1. Стек елементів

На рисунку нижче наведено стек книжок. Єдиною книжкою, до якої ми маємо доступ, так щоб не зруйнувати всю колекцію є верхня книга. Щоб отримати доступ до книжок, що знаходяться нижче, потрібно прибрати по одній книжці зі стопки.

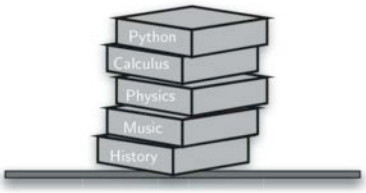


Рисунок 5.2. Стопка книжок утворює стек

Базові дії при роботі зі стеком

Для стеку визначають чотири базових операції:

1. Створення порожнього стеку.
2. Операція `empty()`– визначення, чи є стек порожнім. Повертає булеве значення.
3. Операція `push(item)` – вштовхнути (додати) елемент `item` до стека. Після цього стек змінюється. Елемент `item` стає верхівкою стеку.
4. Операція `pop()` – взяти верхівку стека. Ця операція повертає верхівку стеку. Стек при цьому змінюється.

Крім зазначених вище операцій, опціонально визначають інші операції, наприклад, визначають операцію `top()`, що повертає верхівку стеку, без видалення верхівки зі стеку. Також корисною є операція визначення довжини (тобто кількості елементів) у стеку.

У нижченаведеній таблиці наведено приклад роботи операцій зі стеком. У колонці «Вміст стеку після здійснення операції» напівжирним шрифтом виділено елемент, що є верхівкою стека.

Операція над стеком	Вміст стеку після здійснення операції	Значення, що повертається в результаті здійснення операції
`stack = Stack()`	`[]`	
`stack.empty()`	`[]`	`True`

`stack.push(32)`	`[32]`	
`stack.push(17)`	`[17, 32]`	
`stack.push(26)`	`[26, 17, 32]`	
`len(stack)`	`[26, 17, 32]`	`3`
`stack.empty()`	`[26, 17, 32]`	`False`
`stack.pop()`	`[17, 32]`	`26`
`stack.top()`	`[17, 32]`	`17`

Реалізація стеку на базі вбудованого списку Python

Найпростішу реалізацію стеку у Python можна здійснити на базі вбудованого списку (**`list`**). У такій реалізації елементи стеку розміщують у списку, причому вважається, що верхівка стеку знаходиться в кінці списку. Тоді для додавання та віднімання елементів використовують методи списку **`append()`** та **`pop()`** відповідно.

Опишемо клас `Stack`, що реалізує базові методи роботи зі стеком зазначені у попередньому пункті.

```
class Stack:
    """ Клас, що реалізує стек елементів
        на базі вбудованого  списку Python """

    def __init__(self):
        """ Конструктор """
        self.items = []

    def empty(self):
        """ Перевіряє чи стек порожній

        :return: True, якщо стек порожній
        """
        return len(self.items) == 0

    def push(self, item):
        """ Додає елемент у стек

        :param item: елемент, що додається у стек
        :return: None
        """
        self.items.append(item)

    def pop(self):
        """ Забирає верхівку стека
            Сам елемент при цьому прибирається зі стеку

        :return: Верхівку стеку
        """
        if self.empty():
            raise Exception("Stack: 'pop' applied to empty container")
        return self.items.pop()

    def top(self):
        """ Повертає верхівку стека
            Сам елемент при цьому лишається у стеці

        :return: Верхівку стеку
        """

        if self.empty():
            raise Exception("Stack: 'top' applied to empty container")
        return self.items[-1]

    def __len__(self):
        """ Розмір стеку

        :return: Розмір стеку
        """
        return len(self.items)
```

Для перевірки роботи стеку, створимо новий стек, вштовхнемо туди кілька нових елементів, виштовхнемо елементи зі стеку та виведемо отримані елементи на екран.

```
if __name__ == "__main__":
    stack = Stack()
    stack.push(10)
    stack.push(11)
    stack.push(12)
    stack.push(13)

    print(stack.pop())
    print(stack.pop())
    print(stack.pop())
    print(stack.pop())
```

Результатом роботи програми буде

```
13
12
11
10
```

Реалізація стеку як рекурсивної структури даних

Наведена вище реалізація є дещо штучною, та не відображає самої концепції стеку. Дійсно, при бажанні можна легко модифікувати програму так, щоб доступ був не лише до верхівки стеку, а й до будь-якого його елементу. Крім цього, додавання елементів до стеку може здійснюватися не за сталий час, а за лінійний, наприклад, у випадку, якщо для додавання нового елементу до стеку необхідно провести операцію реалокації[2].

Тому більш правильним є зображення стека у вигляді рекурсивної структури. У такому разі кожен елемент стека є структурою, що крім даних, містить посилання на наступний елемент стека. Доступ є лише до елемента, що є верхівкою стека. Останній елемент у стеку посилається на **None**.

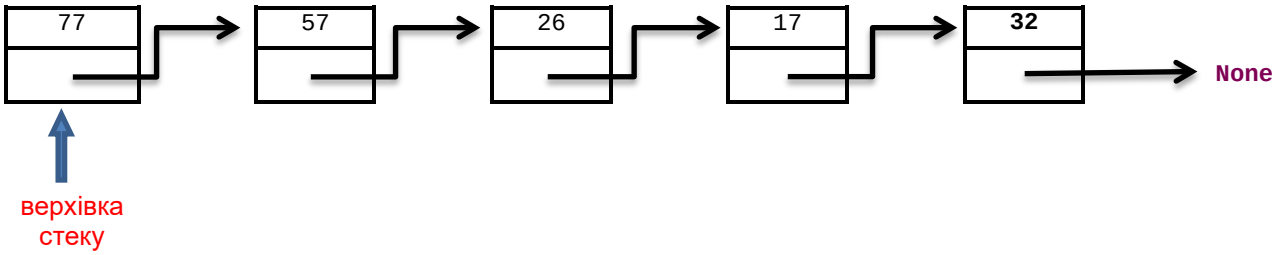


Рисунок 5.3. Рекурсивна реалізація стеку.

Отже, опишемо допоміжний клас `Node` (вузол стека), що містить дані (навантаження вузла) та посилання на наступний вузол стека, тобто об'єкт класу `Node`.

```
class Node:
    """ Допоміжний клас, що реалізує вузол стеку """

    def __init__(self, item):
        """ Конструктор

        :param item: навантаження вузла
        """
        self.item = item    # створюєм поле для зберігання навантаження
        self.next = None    # посилання на наступний вузол стеку
```

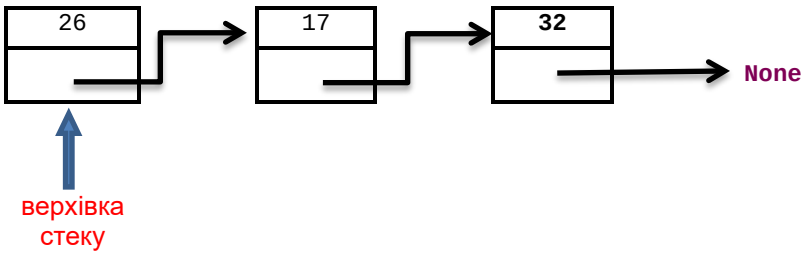
Тоді реалізація класу `Stack` буде мати такий вигляд

```
class Stack:
    """ Клас, що реалізує стек елементів
        як рекурсивну структуру """

    def __init__(self):
        """ Конструктор """
```

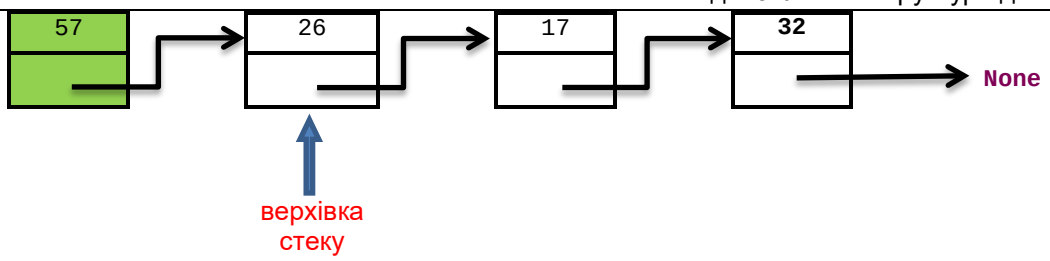
[2] Аллокація (allocation) у програмуванні, процес динамічного виділення пам'яті для розміщення даних об'єктів. Реаллокація (reallocation) – зміна розташування даних об'єкта у пам'яті.

```
        self.top_node = None  # посилання на верхівку стеку

    def empty(self):
        """ Перевіряє чи стек порожній

        :return: True, якщо стек порожній
        """
        return self.top_node is None

    def push(self, item):
        """ Додає елемент у стек

        :param item: елемент, що додається у стек
        """

        new_node = Node(item)               # Створюємо новий вузол стеку
        if not self.empty():                # Якщо стек не порожній, то новий вузол
            new_node.next = self.top_node   # має посилатися на поточну верхівку

        self.top_node = new_node  # змінюємо верхівку стека новим вузлом

    def pop(self):
        """ Забирає верхівку стека
            Сам вузол при цьому прибирається зі стеку

        :return: Навантаження верхівки стеку
        """
        if self.empty():    # Якщо стек порожній
            raise Exception("Stack: 'pop' applied to empty container")

        current_top = self.top_node          # запам'ятовуємо поточну верхівку стека
        item = current_top.item              # запам'ятовуємо навантаження верхівки
        self.top_node = self.top_node.next   # замінюємо верхівку стека наступним вузлом
        del current_top  # видаляємо запам'ятований вузол, що місить попередню верхівку
        return item

    def top(self):
        """ Повертає верхівку стека
            Сам вузол при цьому лишається у стеці

        :return: Навантаження верхівки стеку
        """
        if self.empty():
            raise Exception("Stack: 'top' applied to empty container")
        return self.top_node.item
```

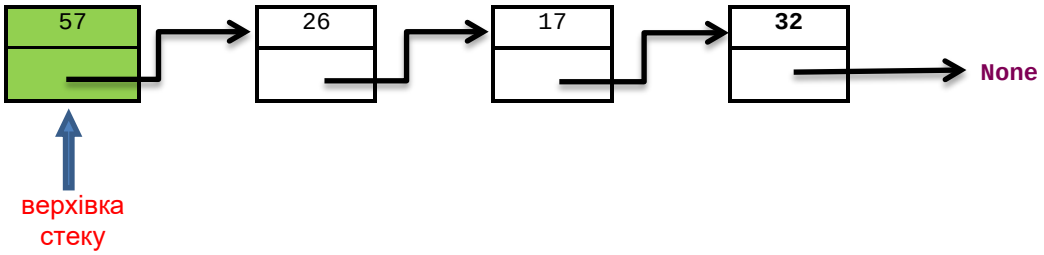
Пояснимо для прикладу схематично роботу методу `push()`. Припустимо у стек додано елементи (32, 17, 26)



Вштовхнемо у стек елемент 57. Для цього створюємо новий вузол стеку, записуємо у нього дані – число 57, а посилання на ступний елемент для нього – верхівка стеку:



Зміщуємо верхівку стека на новий вузол.



Метод `pop()` працює відповідно до метода `push()` у зворотному порядку. Для кращого розуміння реалізації стека як рекурсивної структури, пропонуємо читачу змоделювати роботу метода `pop()` самостійно.

Застосування стеку

Стек у програмуванні має надзвичайно широке застосування, зокрема, у алгоритмах де потрібно дотриматися принципу "останнім прийшов – першим пішов". Напевно чи не найпоширенішим прикладом є інверсія даних, тобто коли послідовність даних потрібно переписати у зворотному порядку. Розглянемо інші класичні застосування стеку.

Збалансовані дужки

Конвертування чисел з однієї системи числення до іншої

Інфіксні та постфіксні арифметичні вирази

Буферне вікно та його застосування

Завдання для самостійної роботи

5.1. Реалізуйте інші операції роботи зі стеком
- Довжина стеку.
- Очищення стеку – видалення всіх елементів.
- Перевірка переповнення стеку. Переповнення стеку, це ситуація, коли кількість елементів досягає критичного (визначеного) значення, після якого додавання нових елементів до стеку не можливе.

5.2. Реалізуйте стек на базі списку, так, щоб вершиною вважався не останній елемент, а **перший**. Порівняйте складність операцій `push` та `pop` обох реалізацій.

5.3. Напишіть програму перевірки чи правильно у заданому арифметичному виразі розставлені дужки.

5.4. Перетворення десяткового числа у двійкову форму.

5.5. Використовуючи стек розв'язати таку задачу: на вхід подається послідовність символів. Необхідно впорядкувати цю послідовність за зростанням. Для впорядкування використайте два стеки.

5.6. В HTML документ формується за допомогою тегів. Кожен тег має дві форми: відкриваючий тег та закриваючий тег. У правильно оформленому HTML документі теги повинні бути збалансованими. Ось приклад простого веб-документа:

```
<html>
   <head>
```

```
      <title>
         Example
      </title>
   </head>

   <body>
      <h1>Hello, world</h1>
   </body>
</html>
```

Напишіть програму, яка перевірить HTML документ на відповідність відкриваючих та закриваючий тегів.

§5.2. Черга

Означення 5.2. **Черга** – динамічна структура даних із принципом доступу до елементів "першим прийшов – першим пішов" (англ. FIFO – first in, first out).

У черги є початок (голова) та кінець (хвіст).

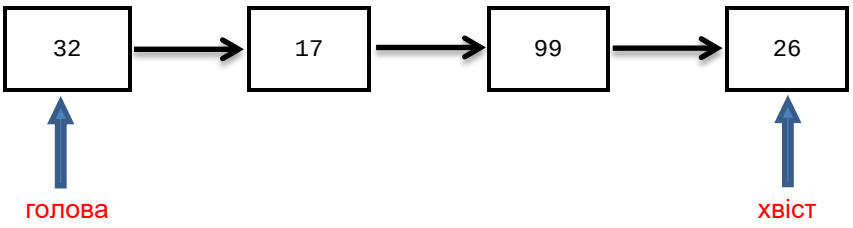


Рисунок 5.4. Черга, має початок та кінець.

Елемент, що додається до черги, опиняється у її хвості.

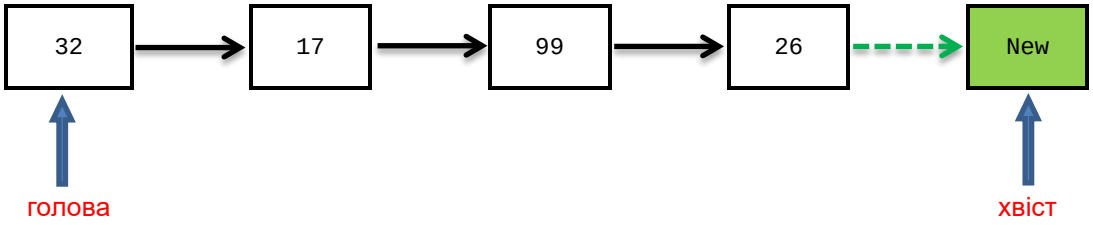


Рисунок 5.5. Додавання елемента «New» в кінець черги

Елемент, що береться (тобто видаляється) з черги, розташований у її голові.

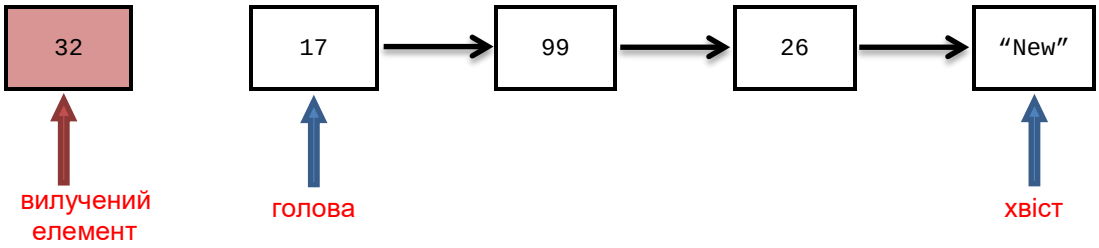


Рисунок 5.6. Вилучення першого елемента «32» з черги

Базові операції для роботи з чергою

Класична реалізація черги вимагає реалізувати структуру даних з такими операціями:

1. Створення порожньої черги.
2. Операція `empty()` – визначення, чи є черга порожньою. Повертає булеве значення.
3. Операція `enqueue(item)` – додати елемент `item` до кінця черги.
4. Операція `dequeue()` – взяти елемент з початку черги.

Як і у випадку зі стеком, крім зазначених вище операцій, опціонально визначають інші операції: перегляд елемента в голові та хвості черги без їхнього видалення з черги, розмір черги, тобто кількість елементів у ній, тощо.

У нижченаведеній таблиці наведено приклад роботи операцій із чергою. У колонці «Вміст черги після здійснення операції» напівжирним шрифтом виділено елемент, що є головою черги, а підкресленням – елемент, що знаходиться у її хвості.

Операція над чергою	Вміст черги після здійснення операції	Значення, що повертається в результаті здійснення операції
`queue = Queue()`	`[]`	
`queue.empty()`	`[]`	`True`
`queue.enqueue(32)`	`[32]`	
`queue.enqueue(17)`	`[32, 17]`	
`queue.enqueue(99)`	`[32, 17, 99]`	
`queue.enqueue(26)`	`[32, 17, 99, 26]`	
`queue.enqueue("New")`	`[32, 17, 99, 26, "New"]`	
`len(queue)`	`[32, 17, 99, 26, "New"]`	5
`queue.empty()`	`[32, 17, 99, 26, "New"]`	`False`
`queue.dequeue()`	`[17, 99, 26, "New"]`	32

Реалізація черги на базі вбудованого списку

Як і у випадку зі стеком, спочатку наведемо простішу реалізацію черги, що базується на вбудованому списку. Опишемо клас `Queue` у якому реалізуємо основні операції, а також функцію визначення розміру черги.

```
class Queue:
    """ Клас, що реалізує чергу елементів
        на базі вбудованого списку Python """

    def __init__(self):
        """ Конструктор """
        self.items = []  # Список елементів черги

    def empty(self):
        """ Перевіряє чи черга порожня

        :return: True, якщо черга порожня
        """
        return len(self.items) == 0

    def enqueue(self, item):
        """ Додає елемент у чергу (у кінець)

        :param item: елемент, що додається
        :return: None
        """
        self.items.append(item)

    def dequeue(self):
        """ Прибирає перший елемент з черги
            Сам елемент при цьому прибирається із черги

        :return: Перший елемент черги
        """
        if self.empty():
            raise Exception("Queue: 'dequeue' applied to empty container")
        return self.items.pop(0)

    def __len__(self):
        """ Розмір черги

        :return: Кількість елементів у черзі
        """
        return len(self.items)
```

Як можна побачити з вищенаведеної реалізації, додавання елементу до черги здійснюється за сталий час $O(1)$. У той час як вилучення елемента з черги здійснюється за час $O(n)$, де n – кількість елементів у черзі. Дійсно, цього вимагає операція `pop(0)` – вилучення першого елементу списку.

Така реалізація не найкращий варіант з практичної точки зору – якщо черга використовується у системі де постійно в черзі знаходиться велика кількість елементів, продуктивність програми буде дуже низькою.

Реалізація черги як рекурсивної структури даних

Наведемо реалізацію черги, у якій операції додавання нового елементу до черги та вилучення елементу з голови здійснюється за сталий час $O(1)$. Така реалізація дуже подібна до рекурсивної реалізації стеку – кожен елемент черги, крім даних, також містить посилання на наступний елемент черги, а для доступу до голови та хвоста черги використовується спеціальний елемент керування.

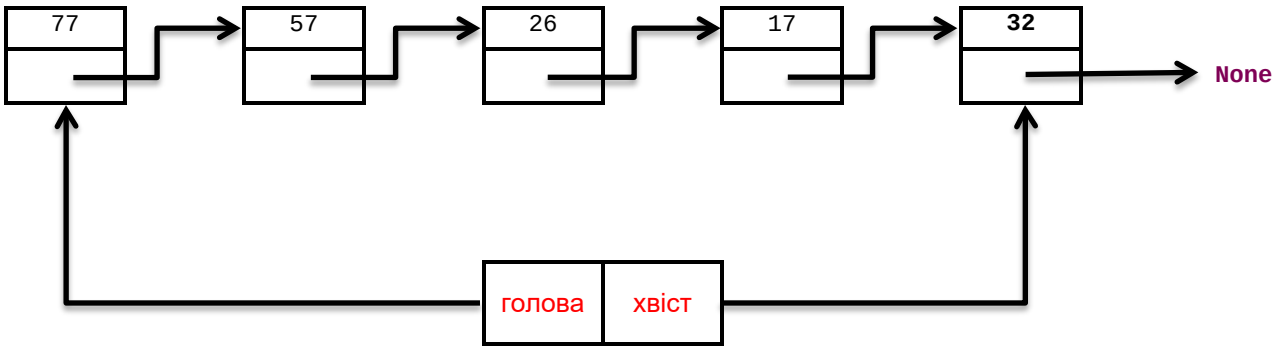


Рисунок 5.7. Рекурсивна реалізація черги.

Як у випадку зі стеком, рекурсивна реалізація черги використовує структуру даних – `Node` (вузол черги), що містить окрім даних посилання на вузол, що містить наступний елемент черги.

```
class Node:
    """ Допоміжний клас - вузол черги.

    Вузлол зберігає у собі навантаження - певну інформаційну частину
    (іншу структуру або тип даних) - та посилання на наступний вузол
    """

    def __init__(self, item):
        """ Конструктор

        :param item: навантаження вузла
        """
        self.item = item    # поле для зберігання навантаження
        self.next = None    # посилання на наступний вузол черги
```

Використовуючи вищеописаний клас опишемо клас `Queue`, що є реалізацією черги як рекурсивної структури даних.

```
class Queue:
    """ Клас, що реалізує чергу елементів
        як рекурсивну структуру """

    def __init__(self):
        """ Конструктор """
        self.front = None  # Посилання на початок черги
        self.back = None   # Посилання на кінець черги

    def empty(self):
        """ Перевіряє чи черга порожня

        :return: True, якщо черга порожня
        """
        # Насправді досить перевіряти лише одне з полів front або back
        return self.front is None and self.back is None

    def enqueue(self, item):
        """ Додає елемент у чергу (в кінець)

        :param item: елемент, що додається
        :return: None
        """
```

```
        new_node = Node(item)       # Створюємо новий вузол черги
        if self.empty():            # Якщо черга порожня
            self.front = new_node   # новий вузол робимо початком черги
        else:
            self.back.next = new_node  # останній вузол черги посилається на новий вузол

        self.back = new_node        # Останній вузол вказує на новий вузол

    def dequeue(self):
        """ Прибирає перший елемент з черги
            Сам елемент при цьому прибирається із черги

        :return: Навантаження голови черги (перший елемент черги)
        """
        if self.empty():
            raise Exception("Queue: 'dequeue' applied to empty container")

        current_front = self.front       # запам'ятовуємо поточну голову черги
        item = current_front.item        # запам'ятовуємо навантаження першого вузла
        self.front = self.front.next     # замінюємо перший вузол наступним
        del current_front                # видаляємо запам'ятований вузол

        if self.front is None:  # Якщо голова черги стала порожньою
            self.back = None    # Черга порожня = хвіст черги теж порожній
        return item
```

Реалізація методів вищеописаного класу дуже подібна до реалізації методів стеку. Зокрема методи `enqueue()` та `dequeue()` концептуально дуже подібні до реалізації методів стеку `push()` та `pop()` відповідно. Проте, звернемо увагу читача на крайові моменти, а саме ситуацію, коли черга порожня у момент додавання нового елементу або стає порожньою внаслідок вилучення елементу. Пропонуємо читачу проаналізувати вищенаведену реалізацію черги та змоделювати процедури додавання та вилучення елементів у черзі.

Завдання для самостійної роботи

5.7. Гра «Лічилка». По колу розташовано n гравців з номерами від 1 до n. У лічилці m слів. Починають лічити з першого гравця. m-й за ліком вибуває. Потім знову лічать з наступного гравця за вибулим. Знову m-й вибуває. Так продовжують, поки не залишиться жодного гравця. Треба показати послідовність номерів, що вибувають, при заданих n та m.

5.8. Симуляція гри «Hot Potato». Реалізуйте вищенаведену програму так, щоб гравці розрізнялися за іменами, а розмір переходу (m) від одного гравця до іншого визначався випадковим чином.

5.9. Симуляція: черга на друк.
(див. http://aliev.me/runestone/BasicDS/SimulationPrintingTasks.html)

5.10. У черзі є m чисел. Проводять n випробувань, унаслідок кожного з яких отримують випадкові числа 0 або 1. Якщо отримано 0, то треба взяти елемент із початку черги й показати його на екрані. Якщо отримано 1, то ввести число з клавіатури та додати до кінця черги. Після завершення випробувань показати залишок черги.

5.11. У магазині стоїть черга з m покупців. Час обслуговування покупця з черги – випадкове ціле число в діапазоні від 1 до t_1. Час додавання нового покупця до черги – випадкове ціле число в діапазоні від 1 до t_2. Промоделюйте стан черги (потрібно показати час виникнення подій – обслуговування та додавання покупця) за період часу T ($T \gg t_1$, $T \gg t_2$).

§5.3. Черга з двома кінцями

Означення 5.3. Черга з двома кінцями (двостороння черга або дек, deque від англ. double ended queue) – динамічна структура даних, елементи якої можуть

- додаватись як у початок (голову), так і в кінець (хвіст);
- вилучатися як з початку, так і з кінця.

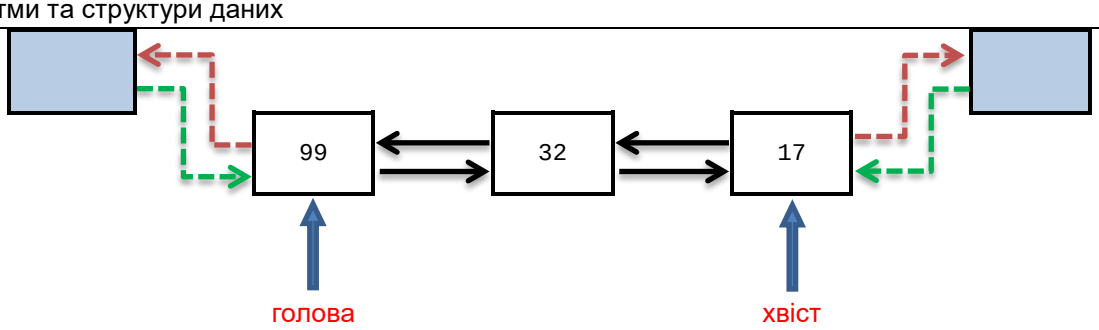


Рисунок 5.8. Дек елементів. Схематично позначено можливість додавання та вилучення елементів як у початок, так і в кінець структури даних.

Базові операції для роботи з деком

Класична реалізація деку вимагає опису структури даних, що підтримує такі операції:
1. Створення порожнього деку.
2. Операція `empty()`– визначення, чи є дек порожнім.
3. Операція `appendleft(item)` – додати елемент `item` до початку дека.
4. Операція `popleft()` – взяти елемент з початку дека.
5. Операція `append(item)` – додати елемент `item` до кінця дека.
6. Операція `pop()` – взяти елемент з кінця дека.

У нижченаведеній таблиці наведено приклад роботи операцій із двосторонньою чергою. У колонці «Вміст деку після здійснення операції» напівжирним шрифтом виділено елемент, що є головою деку, а підкресленням – елемент, що знаходиться у його хвості.

Операція над деком	Вміст деку після здійснення операції	Значення, що повертається в результаті здійснення операції
`deque = Deque()`	`[]`	
`deque.empty()`	`[]`	`True`
`deque.append(32)`	[**32**]	
`deque.append(17)`	[**32**, 17]	
`len(deque)`	[**32**, 17]	2
`deque.empty()`	[**32**, 17]	`False`
`deque.appendleft(99)`	[**99**, 32, 17]	
`deque.appendleft(57)`	[**57**, 99, 32, 17]	
`deque.pop()`	[**57**, 99, 32]	17
`deque.popleft()`	[**99**, 32]	57

Реалізація на базі вбудованого списку

Опишемо клас `Deque`, що є реалізацією деку на базі вбудованого списку.

```
class Deque:
    def __init__(self):
        """ Конструктор деку
        Створює порожній дек.
        """
        self.items = []     # Список елементів деку

    def empty(self):
        """ Перевіряє чи дек порожній

        :return: True, якщо дек порожній
        """
        return len(self.items) == 0

    def append(self, item):
        """ Додає елемент у кінець деку

        :param item: елемент, що додається
        :return: None
```

```
        """
        self.items.append(item)

    def pop(self):
        """ Повертає елемент з кінця деку.

        :return: Останній елемент у деку
        """
        if self.empty():
            raise Exception("Deque: 'popBack' applied to empty container")
        return self.items.pop()

    def appendleft(self, item):
        """ Додає елемент до початку деку

        :param item: елемент, що додається
        :return: None
        """
        self.items.insert(0, item)

    def popleft(self):
        """ Повертає елемент з початку деку.

        :return: Перший елемент у деку
        """
        if self.empty():
            raise Exception("Deque: 'popFront' applied to empty container")
        return self.items.pop(0)

    def __len__(self):
        """ Розмір деку

        :return: Кількість елементів у деку
        """
        return len(self.items)
```

Перевірку роботи деку, описаного вище, можна провести таким чином

```
if __name__ == "__main__":
    D = Deque()        # Створюємо новий дек
    D.append(32)       # Додаємо елемент 32 у кінець деку
    D.append(17)       # Додаємо елемент 17 у кінець деку
    D.appendleft(99)   # Додаємо елемент 99 у початок деку
    D.appendleft(57)   # Додаємо елемент 57 у початок деку

    print(D.pop())      # Виштовхуємо останній елемент (17) з деку
    print(D.popleft())  # Виштовхуємо перший елемент (57) з деку
```

Реалізація двосторонньої черги як рекурсивної структури

Вищенаведена реалізація деку на базі вбудованого списку хоча і є достатньо простою, проте суперечить загальноприйнятому правилу, про те що додаватися та вилучатися елементи з деку мають за сталий час. Це стосується операцій `appendleft()` та `popleft()`, час виконання яких є лінійним за кількістю елементів у деку. Тому, аналогічно до черги, реалізацію деку здійснюють рекурсивним чином.

Як і у випадку зі стеком, та чергою, дек використовує структуру даних – `Node` (вузол деку). Проте, на відміну від вищезгаданих структур, вузол деку, окрім посилання на наступний вузол, містить ще й посилання на попередній вузол.

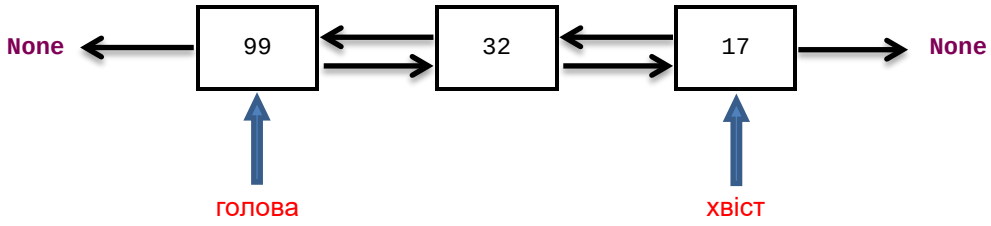
```
class Node:
    """ Допоміжний клас - вузол деку """

    def __init__(self, item):
        """ Конструктор вузла деку

        :param item: Елемент деку
        """
        self.item = item  # поле, що містить елемент деку
        self.next = None  # наступний вузол
        self.prev = None  # попередній вузол
```

Використовуючи цей клас опишемо клас `Deque`, що є реалізацією двосторонньої черги як рекурсивної структури даних.

```
class Deque:
    """ Реалізує дек як рекурсивну структуру. """

    def __init__(self):
        """ Конструктор деку - Створює порожній дек.
        """
        self.front = None  # Посилання на перший елемент деку
        self.back = None   # Посилання на останній елемент деку

    def empty(self):
        """ Перевіряє чи дек порожній

        :return: True, якщо дек порожній
        """
        return self.front is None and self.back is None

    def appendleft(self, item):
        """ Додає елемент до початку деку

        :param item: елемент, що додається
        :return: None
        """
        node = Node(item)           # створюємо новий вузол деку
        node.next = self.front      # наступний вузол для нового - елемент, що є першим
        if not self.empty():        # якщо додаємо до непорожнього деку
            self.front.prev = node  # новий вузол стає попереднім для першого
        else:
            self.back = node  # якщо додаємо до порожнього деку, новий вузол є останнім
        self.front = node     # новий вузол стає першим у деку

    def popleft(self):
        """ Повертає елемент з початку деку.

        :return: Перший елемент у деку
        """
        if self.empty():
            raise Exception('pop_front: Дек порожній')
        node = self.front       # node - перший вузол деку
        item = node.item        # запам'ятовуємо навантаження
        self.front = node.next  # першим стає наступний вузлом деку
        if self.front is None:  # якщо в деку був 1 елемент
            self.back = None    # дек стає порожнім
        else:
            self.front.prev = None  # інакше перший елемент посилається на None
        del node                    # Видаляємо вузол
        return item

    # методи append та pop повністю симетричні appendleft та popleft відповідно
    def append(self, item):
        """ Додає елемент у кінець деку

        :param item: елемент, що додається
        :return: None
        """
        elem = Node(item)
        elem.prev = self.back
        if not self.empty():
            self.back.next = elem
        else:
            self.front = elem
        self.back = elem

    def pop(self):
        """ Повертає елемент з кінця деку.

        :return: Останній елемент у деку
        """
        if self.empty():
            raise Exception('pop_back: Дек порожній')
```

```
        node = self.back
        item = node.item
        self.back = node.prev
        if self.back is None:
            self.front = None
        else:
            self.back.next = None
        del node
        return item

    def __del__(self):
        """ Деструктор - використовується для коректного видалення
            усіх елементів деку у разі видалення самого деку

        :return: None
        """
        while self.front is not None:  # проходимо по всіх елементах деку
            node = self.front  # запам'ятовуємо посилання на елемент
            self.front = self.front.next  # переходимо до наступного елементу
            del node  # видаляємо елемент
        self.back = None
```

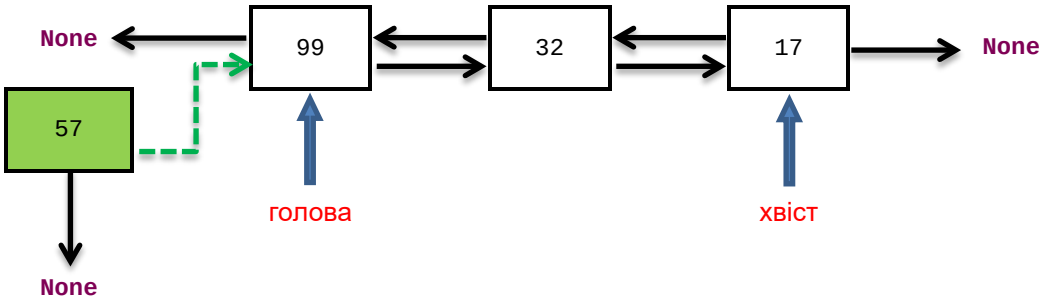
Пояснимо операцію **appendleft()**. Інші операції пропонуємо розібрати читачу самостійно. Отже, розглянемо дек, зображений нижче.



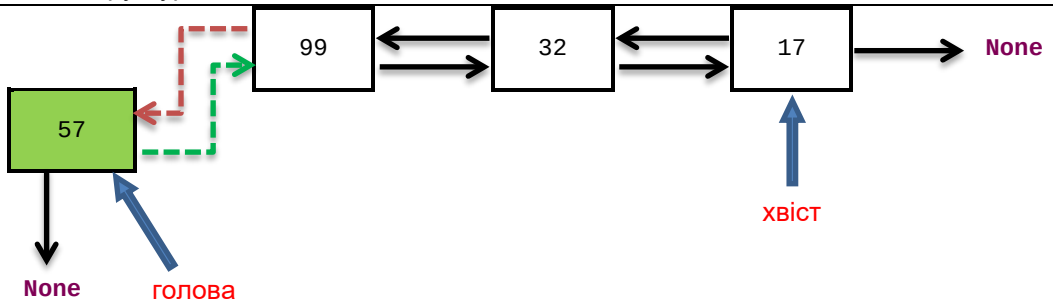
Нехай до нього застосовується операція

```
appendleft(57)  # Додаємо елемент 57 у початок деку
```

Спочатку створюємо новий вузол, у який записуємо число 57, попереднім його вузлом робимо **None**, а наступним – перший елемент деку.



Тепер лишається замінити посилання на попередній елемент для першого елементу деку та змістити голову дека на новий вузол.



Завдання для самостійної роботи

5.12. Як ми бачимо, запропонована реалізація виконує операції `pushFront` та `popFront` за час $O(n)$. Запропонуйте реалізацію деку, так, щоб всі базові операції виконувалися за час $O(1)$.

5.13. Розв'яжіть класичну задачу про «паліндром» з використанням деку.

5.14. Розв'яжіть задачу 5.9, передбачивши чотири можливих результати кожного випробування (випадкові числа від 0 до 3):

1) 0 – взяти елемент із початку дека та показати його на екрані;
2) 1 – увести число з клавіатури та додати його до початку дека;
3) 2 – взяти елемент із кінця дека та показати його на екрані;
4) 3 – увести число з клавіатури та додати його до кінця дека.

5.15. Розв'яжіть вправу 5.10, передбачивши, що через випадковий час від 1 до t_3 до початку черги додається "пільговий" покупець, який обслуговується першим, а через випадковий час від 1 до t_4 не витримує та йде з черги останній покупець.

§5.4. Пріоритетна черга

Елементи в таку чергу додаються у парі з пріоритетом, тобто у вигляді кортежу `(priority, item)`. Ця черга відрізняється від стандартної тим, що елементи вибираються з неї не в порядку якому вони додавалися, а згідно з їхнім пріоритетом. Над пріоритетною чергою допустимі такі операції

1. Створити чергу.
2. Операція **`empty`**`()` – чи порожня черга.
3. Операція **`insert`**`(priority, item)` – вставити пару `(priority, item)` у чергу.
4. Операція **`extract_minimum`**`()` – отримати пару з найвищим пріоритетом. Вважається, що найвищий пріоритет має елемент у якого `priority` є найменшим.

Наведемо приклад реалізації пріоритетної черги, час додавання елементу до якої має порядок $O(1)$, а час отримання елементу $O(n)$.

```
class PriorityQueue:

    def __init__(self):
        """ Конструктор """
        self.items = []    # Список елементів черги,
                           # містить пари (пріоритет, значення)

    def empty(self):
        """ Перевіряє чи чергра порожня

        :return: True, якщо черга порожня
        """
        return len(self.items) == 0

    def insert(self, priority, item):
        """ Додає елемент у чергу разом з його пріоритетом

        :param priority: пріоритет
        :param item: елемент
        :return: None
        """
        self.items.append((priority, item))

    def extract_minimum(self):
        """ Повертає елемент з черги, що має найвищий пріоритет

        :return: елемент з черги з найвищим пріоритетом
```

```
        """
        if self.empty():
            raise Exception("PQueue: 'extract_minimum' applied to empty container")

        # шукаємо елемент з найвищим пріоритетом
        # у нашому випадку, той елемент для якого значення priority є найменшим
        minpos = 0
        for i in range(1, len(self.items)):
            if self.items[minpos][0] > self.items[i][0]:
                minpos = i

        return self.items.pop(minpos)[1]
```

Зауваження. Реалізація пріоритетної черги, наведена вище, скоріше має ознайомчий характер. Хоча така реалізація є достатньою для розв'язання багатьох прикладних задач, проте, у випадку, великої кількості даних, продуктивність черги може бути незадовільною. Пізніше ми розглянемо реалізацію пріоритетної черги на базі структури даних Купа. Така реалізація дозволяє здійснювати вставку та отримання елементів з черги за логарифмічний час.

§5.5. Списки

Однозв'язний список

Списки відрізняються від стеків, черг та деків тим, що мають можливість безліч разів проходити вздовж списку, отримувати доступ до будь-якого елемента, не змінюючи сам список. Існує декілька різновидів списків: однозв'язні списки, кільцеві списки, двозв'язні списки.

Означення 5.4. **Класичний** (**зв'язний або однозв'язний**) **список** – це динамічна структура даних, що складається з елементів (як правило одного типу), пов'язаних між собою у строго визначеному порядку: кожен елемент списку вказує на наступний елемент списку.

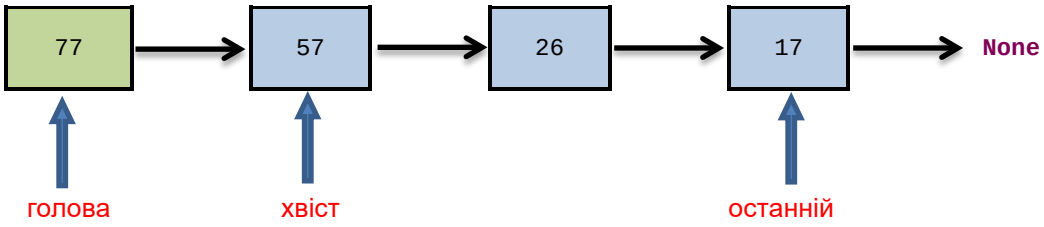


Рисунок 5.9. Класичний список

Елемент, на який немає посилання називається **початковим** або **головою** списку. Останній елемент не посилається на жоден елемент списку (тобто посилається на `None`). **Хвостом** списку будемо називати список, що складається з усіх елементів вихідного списку, крім першого. Фактично хвіст списку це посилання на другий елемент цього списку.

Як бачимо, організація даних однозв'язного списку дуже подібна на таку для черги або стеку з рекурсивною реалізацією. Відмінністю списків від вищезгаданих структур є операції які проводять над ними. У однозв'язному списку можна пересуватися лише від початку в сторону його кінця, використовуючи операцію отримання хвоста списку.

Базовий набір дій над класичним списком:

Отже, базовий набір дій над однозв'язним списком є таким:

1. Створити список.
2. Операція визначення, чи порожній список.
3. Додати елемент у початок списку.
4. Взяти перший елемент списку (без зміни всього списку).
5. Отримати хвіст списку.

Реалізація зв'язного списку у мові Python

http://aliev.me/runestone/BasicDS/ImplementinganUnorderedListLinkedLists.html

Список із поточним елементом

Означення 5.5. **Список із поточним елементом** – різновид класичного списку – динамічна структура даних, що складається з елементів (як правило одного типу), пов'язаних між собою, і структури керування, що вказує на поточний елемент структури.

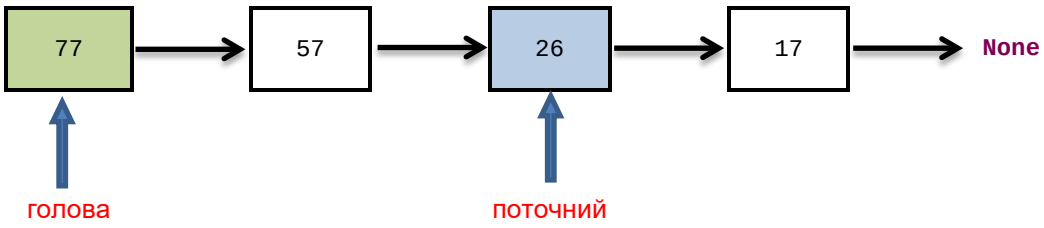


Рисунок 5.10. Список з поточним елементом

Аналогічно до однозв'язного списку, у списку з поточним елементом можна пересуватися лише у одному напрямку – під початку до кінця списку. Проте на відміну від однозв'язного списку, де така операція здійснювалася через операцію визначення хвоста списку, тут використовується поточний елемент.

Базовий набір дій над списками з поточним елементом:

1. Почати роботу.
2. Операція визначення, чи порожній список.
3. Зробити поточним перший елемент списку.
4. Перейти до наступного елемента.
5. Отримати поточний елемент (список при цьому не змінюється).
6. Вставити новий елемент у список перед поточним (або після поточного).
7. Видалити поточний елемент у списку.

Реалізація списку з поточним елементом у мові Python

Реалізуємо допоміжний клас `Element`, що буде мати два поля, перше з яких буде містити дані, а друге посилання на елемент такого ж типу `Element`.

```
class Element:
    '''Реалізує елемент списку.'''

    def __init__(self, data):
        '''Створити елемент.'''
        self._data = data  # дані, що зберігаються у елементі списку
        self._next = None  # посилання на наступний елемент списку

    def getData(self):
        return self._data

    def setData(self, newdata):
        self._data = newdata

    def getNext(self):
        return self._next

    def setNext(self, newnext):
        self._next = newnext
```

За допомогою, цього класу реалізуємо список з поточним елементом. Через таке зображення списку, тобто, через елементи, кожен з яких має посилання на інші елементи, такі структури даних називають рекурсивними.

```
class ListWithCurrent:

    def __init__(self):
        '''Створити порожній список.
        '''
        self._head = None # Перший елемент списку
        self._curr = None # Поточний елемент списку
```

```
        self._prev = None # Елемент, що передує поточному
                          # елементу списку

    def empty(self):
        '''Чи порожній список?.'''
        return self._head == None

    def reset(self):
        '''Зробити поточний елемент першим.'''
        self._curr = self._head
        self._prev = None

    def next(self):
        '''Перейти до наступного елемента.'''
        if self._curr != None:
            self._prev = self._curr
            self._curr = self._curr.getNext()
        else:
            raise StopIteration

    def getCurrent(self):
        '''Отримати поточний елемент .'''
        if self._curr != None:
            return self._curr.getData()
        else:
            return None

    def insert(self, data):
        "Вставити новий елемент у список перед поточним "
        elem = Element(data)
        elem.setNext(self._curr)

        if self._curr == self._head:
            self._head = elem

        if self._prev != None:
            self._prev.setNext(elem)

        self._prev = elem

    def remove(self):
        "Видалити поточний елемент у списку"
        pass # TODO: Implement by yourself
```

Отже, тепер, для створення списку скористаємося командою:

```
l = ListWithCurrent()
```

а для додавання елементів, відповідно

```
l.insert(11)
l.insert(12)
l.insert(13)
l.insert(14)
l.insert(15)
l.insert(16)
```

Для зручності перевизначимо у нашому класу «магічний» метод, для зручного доступу до даних елементів, а саме

```
    def __str__(self):
        return str(self.getCurrent())
```

Тепер, для виведення поточного елементу досить скористатися командою

```
    print(l)
```

Оскільки список, це колекція, додамо до опису нашого класу методи, що надають йому ітераційний протокол

```
    def __iter__(self):
        self._iterator = self._head   # створюємо змінну-ітератор
        return self

    def __next__(self):
        if self._iterator != None:
            cur = self._iterator.getData()
            self._iterator = self._iterator.getNext()
            return cur
        else:
            raise StopIteration
```

Після цього ми можемо скористатися звичайним циклом по колекції для виведення всіх елементів списку

```
for el in l:
    print(el)
```

або що те ж саме

```
it = iter(l)
while True:
    try:
        print(next(l))
    except StopIteration:
        break
```

Кільцевий список

Означення 5.6. **Кільцевий список** – різновид списку з поточним елементом, для якого не визначено перший та останній елементи. Усі елементи зв'язані у кільце, відомий лише порядок слідування.

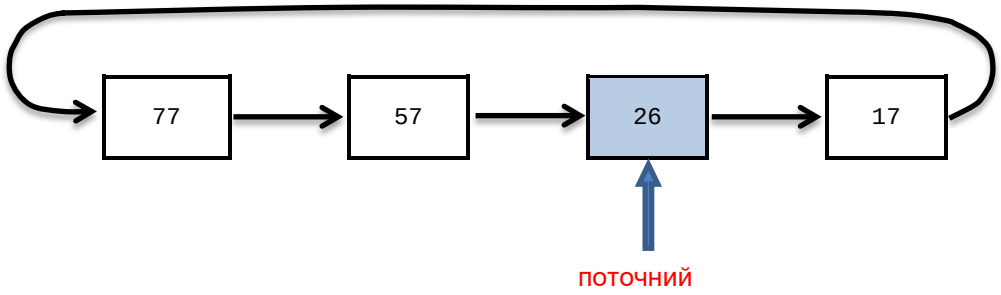


Рисунок 5.11. Кільцевий список

Базовий набір дій над кільцевими списками:

1. Почати роботу.
2. Перейти до наступного елемента.
3. Отримати поточний елемент (список при цьому не змінюється).
4. Вставити новий елемент у список перед поточним.
5. Видалити поточний елемент у списку.

Двозв'язний список

Означення 5.7. **Двобічно зв'язаний** (двозв'язний) **список** – динамічна структура даних, що складається з елементів одного типу, зв'язаних між собою у строго визначеному порядку. При цьому визначено перший та останній елементи у списку, а кожен елемент списку вказує на наступний і попередній елементи у списку. Перший елемент має попереднім елементом посилання на невизначений елемент `None`. Аналогічно, `None` буде наступним елементом для останнього елементу списку.

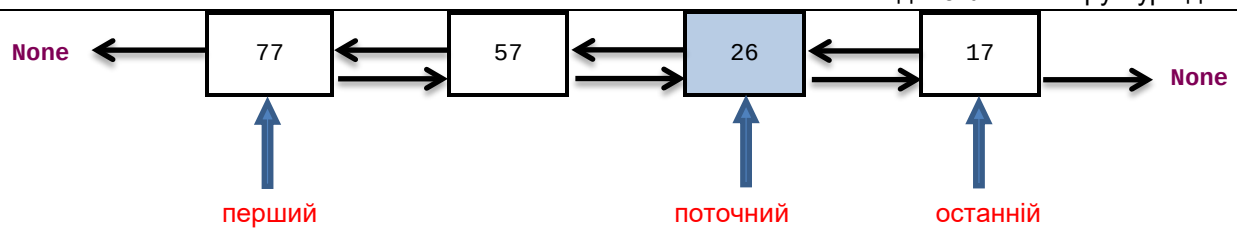


Рисунок 5.12. Двозв'язний список

Додатково для двозв’язного списку визначається поточний елемент. Рух по списку здійснюється за рахунок пересування поточного елементу на попередній або наступний елемент списку.

Базовий набір дій над двозв'язними списками:

1. Створити список.
2. Операція `empty()` – визначення, чи порожній список. Повертає булеве значення.
3. Зробити поточними перший елемент списку.
4. Зробити поточними останній елемент списку.
5. Перейти до наступного елемента.
6. Перейти до попереднього елемента.
7. Отримати поточний елемент.
8. Вставити новий елемент перед поточним.
9. Вставити новий елемент після поточного.
10. Видалити поточний елемент.

Завдання для самостійної роботи

5.16. Запропонуйте реалізацію для кожного з визначених вище списків у мові програмування Python.

5.17. Для кожного з типів списків реалізуйте алгоритми визначення розміру списку та лінійного пошуку у списку.

5.18. Реалізуйте стек, використовуючи зв’язний список.

5.19. Реалізуйте чергу, використовуючи зв’язний список.

5.20. Продумайте і проведіть експеримент, що порівнює продуктивності списків Python і зв’язних списків.

5.21. Продумайте і проведіть експеримент, що порівнює продуктивність стека і черги, реалізованих на базі вбудованих списків Python, з реалізаціями на основі зв'язаного списку.

5.22. Для кожного зі списків реалізуйте ітераційний протокол.

РОЗДІЛ 6. ДЕРЕВА

§6.1. Означення, приклади та реалізація у Python

§6.2. Алгоритми на деревах

DFS та BFS для дерев

Префіксне дерево

Побудова каркасного дерева та алгоритм Прима

§6.3. Бінарні дерева

Бінарне дерево пошуку

Збалансовані дерева пошуку

АВЛ-дерево

Червоно-чорне деремо

Двійкова купа та пріоритетна черга

Алгоритм сортування на базі купи

РОЗДІЛ 7. ТЕОРІЯ ГРАФІВ

§7.1. Означення, приклади та реалізація у Python

Означення

Графи це математичні модель, що зображає сукупність об’єктів та зв’язки між ними. Велика кількість структур, які мають практичну цінність в математиці, інформатиці та інших науках, можуть бути зображені як графи. Наприклад, системи доріг, схема руху метрополітену, множина аеропортів та регулярні рейси літаків між ними, комунікації в інтернеті, тощо можуть бути зображені у вигляді графів.

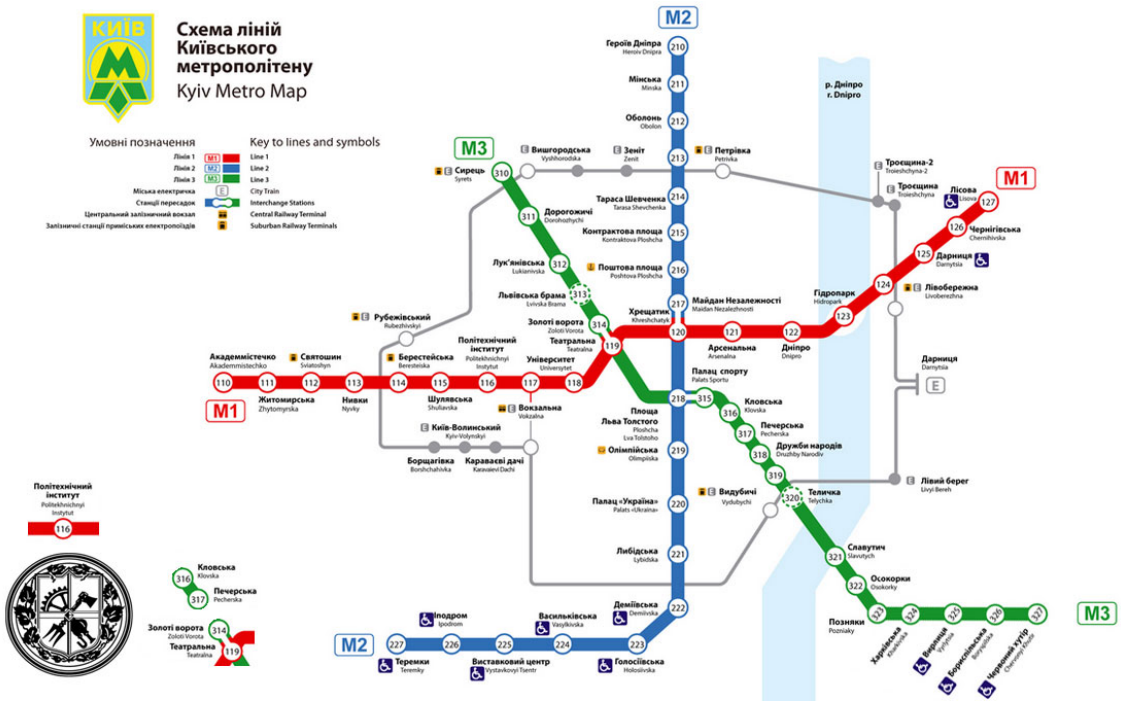


Рисунок 1

Неформально граф – це набір точок (вершин) і ліній (ребер), що ці точки з’єднують. Отже, наведемо строге означення графу.

Означення 7.1. Графом G називається пара

$$G = (V,E)$$

де V – множина вершин графу, E – множина ребер графу.

Вершина (або вузол) графу – це математична абстракція, яка моделює об’єкти. Наприклад, на зазначеній вище схемі – вершинами є станції метро. Вершина, як правило, має ім’я, яке називається *ключем* або ідентифікатором вершини.

$$\{v_1, v_2, \dots, v_n\}$$

Ключі використовуються, для того щоб розрізняти вершини. На схемі вище, ключами є назви станцій метро. Часто при розв’язанні конкретних задач виникає необхідність пов’язати з певною вершиною деякі дані. Ці дані називаються корисним **навантаженням вершини** (або просто навантаженням вершини). У нашому прикладі навантаженням може бути час роботи станції, визначні місця поруч, тощо.

Ребра (або дуги) графу визначають відношення (тобто з’єднання) між парами вершин. Так у нашому прикладі схеми метрополітену, ребрами будуть відрізки, що сполучають сусідні станції метро. Ребра визначаються парою (тобто кортежем) вершин які вони з’єднують.

Кількість вершин у графі називається **порядком графа**, а кількість його ребер – **розміром графу**.

Нижче наведено граф, що має 6 вершин (порядок графа 6)

$$V = \{v0, v1, \dots, v5\}$$

та 9 ребер (розмір графа – 9)

$$E = \{(v0, v1), (v1, v2), (v2, v3), (v3, v4), (v4, v0), (v0, v5), (v5, v4), (v3, v5), (v5, v2)\}$$

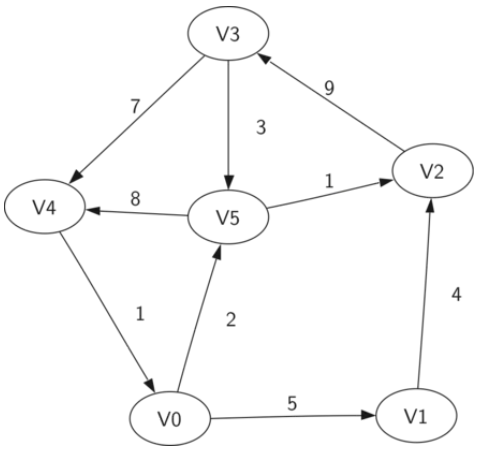


Рисунок 2

Ребра графу можуть бути **одно**- або **двонаправленими**. Однонаправлене ребро (також використовується термін дуга) графу вказує на односторонній зв'язок пари вершин графу. Прикладами дуг, тобто однонаправлених ребер в реальному житті можуть бути вулиці з одностороннім рухом, що з'єднують перехрестя або транспортні розв'язки (тобто вершини). Дуги позначаються на малюнку відрізками зі стрілками, що визначають дозволений рух від однієї вершини до іншої. На рисунку вище всі ребра є дугами. Граф, всі ребра якого є дугами, називається **орієнтованим**. Наведений вище на рисунку 2 граф є орієнтованим.

Якщо ребро двонаправлене (у подальшому просто ребро), то рух по ньому між двома вершинами може здійснюватися у обох напрямках. Наприклад, на нашій схемі метрополітену, всі ребра (тобто відрізки, що сполучають станції метро - вузли) є двонаправленими. На схемах, ребра позначаються відрізками, що сполучають вершини.

Граф всі ребра якого двонаправлені називається **неорієнтованим** графом. Схема київського метрополітену, наведена вище утворює неорієнтований граф. Нижче наведено приклад іншого неорієнтованого графу.

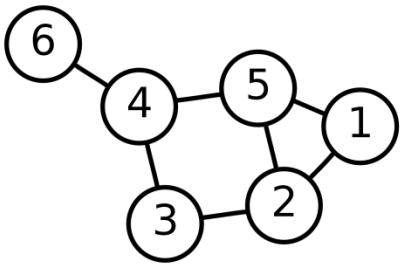


Рисунок 3

У неорієнтованих графах вершини сполучені ребром називають **суміжними** (або сусідами). Також називають **суміжними ребра**, що мають спільну вершину. Вершина 6 графу зображеного на рисунку 3 має суміжну вершину 4, а вершина 4, має три суміжних вершини 3, 5 та 6.

Для орієнтованих графів, суміжними з вершиною v називаються вершини, що з'єднуються з вершиною v, дугою (з початком у точці v). Так, для прикладу, у графі зображеному на рисунку 2, вершини $V1$ та $V5$ є суміжними з вершиною $V0$. Протилежне твердження не правильне.

Степінь вершини у неорієнтованому графі це кількість ребер, що виходить з цієї вершини. Степінь вершини v позначається

$$deg\, v$$

Для графу, зображеного на рисунку 3

$$deg\, 6 = 1$$
$$deg\, 4 = 3$$

Очевидно, що для неорієнтованих графів степінь вершини дорівнює кількості її сусідів.

Напівстепінь входу вершини v орієнтованого графа – це кількість дуг, які входять у дану вершину. Позначається

$$deg^+ v$$

Напівстепінь виходу вершини v графа – це кількість дуг, які виходять з даної вершини.

$$deg^- v$$

Вершина з напівстепенем входу 0 називається **джерелом**, а вершина з напівстепенем виходу 0 – **стоком**.

Ребра можуть мати **вагу**. Вага визначає «вартість» переміщення від однієї вершини до іншої. Наприклад, у графі, що моделює схему доріг, що зв'язує міста, вага може визначати відстань між двома містами. На рисунках, вагу вказують над відповідними ребрами (див. Рисунок 2). Граф, ребра якого мають вагу називається **зваженим**.

Шлях

Маршрутом в графі називається послідовність вершин і ребер, яка має такі властивості:

1. вона починається і закінчується вершиною;
2. вершини і ребра в ній чергуються;
3. будь-яке ребро цієї послідовності має своїми кінцями дві вершини: що безпосередньо передує йому в цій послідовності і наступну що йде відразу за ним.

Прикладом маршруту у графі зображеному на рисунку може бути 2

$$v0 \to v1 \to v2 \to v3$$

Перша і остання вершини в цій послідовності називаються початком і кінцем маршруту. Вищезначений маршрут, що веде з точки $v0$ у точку $v3$ будемо позначати.

$$v0 \rightsquigarrow v3$$

Шляхом називається такий маршрут, в якому жодне ребро не зустрічається двічі. Отже, формально, шлях з вершини w_1 у вершину w_n можна визначити як

$$w_1, w_2, \ldots, w_n$$

такий, що $(w_i, w_{i+1}) \in E$, для всіх $1 \le i \le n-1$.

Довжиною шляху у незваженому графі називається кількість ребер у цьому шляху. Зважений шлях у (зваженому) графі має довжину, що є сумою всіх ваг ребер, що входять до цього шляху. Шлях з вершини у себе в неорієнтованому графі має довжину 0.

Шлях називається **замкненим** або **циклічним**, якщо він починається та закінчується у одній і тій же вершині.

Не замкнений шлях називається простим, якщо кожна вершина у ньому відвідується лише раз. Замкнений шлях називається простим, якщо кожна вершина, за виключенням початкової та кінцевої вершини у ньому відвідується лише один раз. Простий циклічний шлях називається **циклом**. Для циклу не є принциповим є яка вершина є його початком і кінцем.

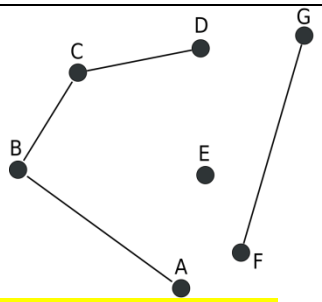
Граф, що не містить циклів називається **ациклічним**.

Кажуть, що вершина w_1 досяжна з вершини w_0, якщо існує шлях з вершини w_0 у вершину w_1

Зв'язність графів

Не орієнтований граф називається **зв'язним**, якщо у ньому будь-яка вершина є досяжною з будь-якої іншої. Граф на рисунку 3 є зв'язним. Неорієнтований граф називається **не зв'язним**, якщо він не є зв'язним.

Бінарне відношення на множині вершин графа, задане як вершина w_1 досяжна з вершини w_0 є відношенням еквівалентності і відповідно розбиває граф на класи еквівалентності, що називаються **компонентами зв'язності**. Очевидно, що якщо граф має лише одну компоненту зв'язності, то він зв'язний (обернене твердження також є правильним). Рисунок нижче демонструє граф, що має три компоненти зв'язності.



Для орієнтованих графів поняття зв'язності не є коректним.

Реалізація графу на мові Python

Існує два широковідомих підходи для реалізації графів:
- Матриця суміжності
- Список суміжності

Матриця суміжності

Одним з найпростіших способів реалізації графу є використання матриці суміжності – двовимірної матриці, у якій i-й рядок відповідає вершині графа v_i, j-й стовпчик відповідає вершині графа v_j, а елемент матриці на перетині i-го рядка та j-го стовпчика показує, чи існує ребро з вершини v_i до вершини v_j. Нульовий елемент матриці суміжності говорить, що ребро між відповідними вершинами відсутнє. Ненульове значення вказує, що існує ребро з вершини v_i до вершини v_j, причому це значення визначає вагу ребра (v_i, v_j). Для незважених графів цю вагу, як правило встановлюють 1.

На рисунку 4 показана матриця суміжності для графа з рисунку 2.

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

Рисунок 4

Переваги:
- Простота реалізації
- Математична наочність
- Однаково реалізується як зважений так і не зважений граф.

Недоліки
- Проблематично додавати/видаляти вершини
- Великий об'єм оперативної пам'яті, що використовується даремно
- Для ідентифікації вершин (без додаткового виділення коду) можуть використовуватися лише послідовні натуральні числа
- Навантаження вершин треба реалізовувати окремою структурою даних.

Отже, матрицю суміжності для графів рекомендується використовувати для задач, у яких кількість вершин є не значною (максимум 2к вершин), кількість вершин наперед відома та задача не передбачає додавання чи видалення вершин.

Наведемо реалізацію графа з використанням матриці суміжності. Клас `Graph` містить лише конструктор та метод для зв'язування двох вершин ребром.

```
class Graph:
    def __init__(self, isOriented=False, aInitialVertexNumber=20):
        self.mIsOriented = isOriented             # Поле чи орієнтований граф
        self.mVertexNumber = aInitialVertexNumber # Лічильник вершин у графі

        # Створюємо матрицю суміжності заповнену нулями
        self.mAdjacentMatrix = []
        for i in range(self.mVertexNumber):
            self.mAdjacentMatrix.append([0] * self.mVertexNumber)

    def addEdge(self, fromVert, toVert, weight=1):
        "Додавання ребра з кінцями в точках fromVert та toVert з вагою weight"
        assert 0 <= fromVert < self.mVertexNumber
        assert 0 <= toVert < self.mVertexNumber
        self.mAdjacentMatrix[fromVert][toVert] = weight

        # Якщо граф є неорієнтованим, то треба встановити зворотній
        # зв'язок з вершини toVert до fromVert
        if not self.mIsOriented:
            self.mAdjacentMatrix[toVert][fromVert] = weight
```

Щоб створити граф, зображений на рисунку рисунку 2 скористаємося ланцюжком команд

```
g = Graph(True, 6)  # Створюємо орієнтований граф

# Додаємо ребра та встановлюємо для кожного з них вагу
g.addEdge(0, 1, 5)
g.addEdge(0, 5, 2)
g.addEdge(1, 2, 4)
g.addEdge(2, 3, 9)
g.addEdge(3, 4, 7)
g.addEdge(3, 5, 3)
g.addEdge(4, 0, 1)
g.addEdge(5, 4, 8)
g.addEdge(5, 2, 1)
```

Список суміжності

Оптимальнішим та гнучкішим способом реалізації розрідженого графа є використання списку суміжності. У такому зображенні граф фактично є списком усіх вершин, кожна з яких володіє інформацією про пов'язані з нею вершини.

На рисунку 5 показаний список суміжності для графа з рисунку 2.

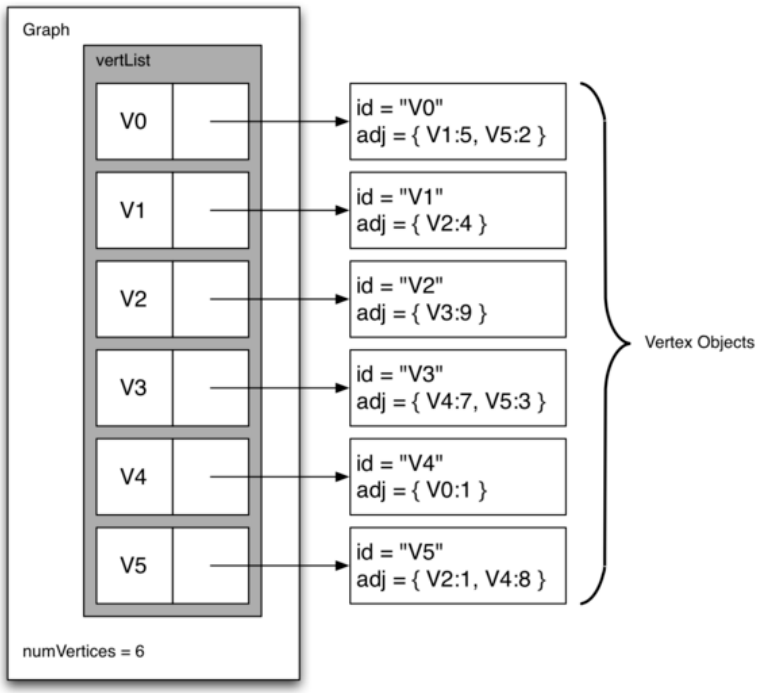


Рисунок 5

Перевагою такої реалізації є те, що вона дозволяє компактно зображувати розріджені графи. Також в списку суміжності легко знайти всі посилання, безпосередньо пов'язані з конкретною вершиною.

Реалізація списку суміжності буде складатися з двох класів – класу `GraphVertex`, для моделювання вершин графу, та класу `Graph`, який власне буде списком вершин графу.

Для зручності реалізуємо клас `GraphVertex` як нащадок класу `Vertex`, що містить основні атрибути та методи для роботи з вершинами, такі як ідентифікатор вершини, її навантаження тощо.

```
class Vertex:
    def __init__(self, vertexId):
        self.mId = vertexId    # Ідентифікатор вершини
        self.mData = None      # Навантаження (дані) вершини

    def getId(self):
        "Отримати ідентифікатор вершини"
        return self.mId

    def setData(self, aData):
        "Встановити навантаження на вершину"
        self.mData = aData

    def getData(self):
        "Отримати навантаження на вершину"
        return self.mData

    def __str__(self):
        "Зображення вершини у вигляді рядка"
        return str(self.mId) + ": Data=" + str(self.getData())
```

Клас `GraphVertex` буде розширяти клас `Vertex`, додаючи всі необхідні атрибути та методи, що необхідні будуть для роботи з графами. Тут ми додамо поле `mNeighbors`, яке буде містити список сусідів вершини у вигляді словника, де ключем буде ключ вершини-сусіда, а значенням – значення ваги ребра (дуги у орієнтованому графі), що з'єднує вершину з сусідом. Наприклад, якщо словник `mNeighbors` буде мати вигляд,

```
{17: 4, 3: 2}
```

то означатиме, що наша вершина має сусідами вершини 17 та 3 з вагами ребер 4 та 2 відповідно.

```
class GraphVertex(Vertex):
    def __init__(self, vertexId):
        super().__init__(vertexId)  # Викликаємо конструктор батьківського класу
```

```
        self.mNeighbors = {}        # Список сусідів вершини у вигляді пар
                                    # (ім'я_сусіда: вага_ребра)

    def addNeighbor(self, aVertex, aWeight=1):
        """Додати сусіда, тобто ребро, що сполучає
           поточну вершину з вершиною aVertex з вагою aWeight """
        if isinstance(aVertex, Vertex):  # Якщо aVertex - вершина
            self.mNeighbors[aVertex.getId()] = aWeight
        else:                            # Якщо aVertex - ім'я (ключ) вершини
            self.mNeighbors[aVertex] = aWeight

    def getNeighbors(self):
        """Повернути список всіх сусідів поточної вершини"""
        return self.mNeighbors

    def getWeight(self, aNeighbor):
        "Отримати вагу ребра, що сполучає поточну вершину сусідом aNeighbor"
        if isinstance(aNeighbor, Vertex):  # Якщо aNeighbor - вершина (не ім'я)
            return self.mNeighbors[aNeighbor.getId()]
        else:  # Якщо aNeighbor - ім'я (ключ) сусідньої вершини
            return self.mNeighbors[aNeighbor]

    def __str__(self):
        "Зображення вершини у вигляді рядка у разом з усіма її сусідами"
        return super().__str__() + ' connected to: ' + str(self.mNeighbors)
```

Клас `Graph` буде використовувати клас `GraphVertex`, тому, якщо він реалізований у іншому модулі, на початку програми потрібно не забути імпортувати його. Імпорт може виглядати інакше в залежності від розташування модуля у проекті.

```
class Graph:
    """ Граф, що задається списком суміжних вершин """

    def __init__(self, isOriented=False):
        self.mIsOriented = isOriented  # Поле чи орієнтований граф
        self.mVertexNumber = 0         # Лічильник вершин у графі
        self.mVertices = {}            # Список (словник) вершин у графі у
вигляді пар (ключ, вершина)

    def addVertex(self, aVertex):
        """ Додає вершину у граф, якщо така вершина не міститься у ньому """
        if aVertex in self:  # Якщо вершина міститься у графі, її вже не треба
додавати
            return False

        if not isinstance(aVertex, GraphVertex):  # Якщо aVertex - вершина (не
ім'я)
            newVertex = GraphVertex(aVertex)      # створюємо нову вершину з
іменем aVertex
            self.mVertices[aVertex] = newVertex   # додаємо цю вершину до списку
вершин графу
        else:                                         # Якщо aVertex - ім'я
ключ вершини
            self.mVertices[aVertex.getId()] = aVertex  # додаємо цю вершину до
списку вершин графу

        self.mVertexNumber += 1                   # Збільшуємо лічильник вершин у
графі
        return True

    def getVertex(self, aVertex):
        "Повертає вершину графу, якщо така вершина міститься у графі"
        assert aVertex in self

        # Визначаємо ключ вершини
```

```
        key = aVertex.getId() if isinstance(aVertex, GraphVertex) else aVertex
        return self.mVertices[key]

    def getVertices(self):
        "Повертає список всіх вершин у графі"
        return self.mVertices

    def addEdge(self, fromVert, toVert, weight=1):
        "Додавання ребра з кінцями в точках fromVert та toVert з вагою weight"
        if fromVert not in self:      # якщо вершина fromVert ще не міститься у
графі
            self.addVertex(fromVert)  # Додаємо її
        if toVert not in self:        # якщо вершина toVert ще не міститься у
графі
            self.addVertex(toVert)    # Додаємо її

        # встановлюємо зв'язок (тобто ребро) між вершинами fromVert та toVert
        self[fromVert].addNeighbor(toVert, weight)

        if not self.mIsOriented:  # Якщо граф не орієнтований, то треба додати
зворотній зв'язок
            self.mVertices[toVert].addNeighbor(fromVert, weight)

    def setVertexData(self, aVertex, aData):
        "Встановлення навантаження aData на вершину з іменем aVertex"
        assert aVertex in self # Перевірка чи міститься вершина в графі
        self[aVertex].setData(aData)

    def getVertexData(self, aVertex):
        "Повернення навантаження вершини з іменем aVertex"
        assert aVertex in self  # Перевірка чи міститься вершина в графі
        return self[aVertex].getData()

    def __contains__(self, aVertex):
        "Перевірка чи міститься вершина з іменем vertex у графі"
        if isinstance(aVertex, GraphVertex):  # Якщо aVertex - вершина (не ім'я)
            return aVertex.getId() in self.mVertices
        else:                                 # Якщо aVertex - ім'я (ключ) вершини
            return aVertex in self.mVertices

    def __iter__(self):
        "Ітератор для послідовного проходження всіх вершин у графі"
        return iter(self.mVertices.values())

    def __len__(self):
        "Перевизначення методу len() як кількість вершин у графі"
        return self.mVertexNumber

    def __str__(self):
        "Зображення графа разом з усіма вершинами і ребрами у вигляді рядка"
        s = ""
        for vertex in self:
            s = s + str(vertex) + "\n"
        return s

    def __getitem__(self, vertex):
        return self.getVertex(vertex)
```

Щоб створити граф, зображений на рисунку рисунку 2 скористаємося ланцюжком команд

```
g = Graph(True)  # Створюємо орієнтований граф

# Додаємо ребра за ідентифікаторами вершин
g.addEdge(0, 1)
g.addEdge(0, 5)
```

```
g.addEdge(1, 2)
g.addEdge(2, 3)
g.addEdge(3, 4)
g.addEdge(3, 5)
g.addEdge(4, 0)
g.addEdge(5, 4)
g.addEdge(5, 2)

v6 = GraphVertex(6)
v7 = GraphVertex(7)


g.addVertex(v6)
g.addVertex(v7)

# Додаємо ребра по вершинах
g.addEdge(v6, v7)
g.addEdge(v7, 5)
```

Завдання для самостійної роботи

7.1. Модифікуйте вищенаведений клас Graph що реалізує граф як список суміжності, так, щоб з його допомогою можна було реалізовувати зважений граф.

7.2. Створіть програму, що знаходить та виводить степені всіх вершин незваженого графу. Реалізуйте два варіанти програми – через матрицю суміжності та списки суміжності.

7.3. Задано орієнтований граф. Напишіть програму, яка визначає чи має граф джерела та стоки.

7.4. Задано орієнтований граф та шлях, що описується послідовністю вершин. Потрібно перевірити, чи належить цей шлях заданому графу.

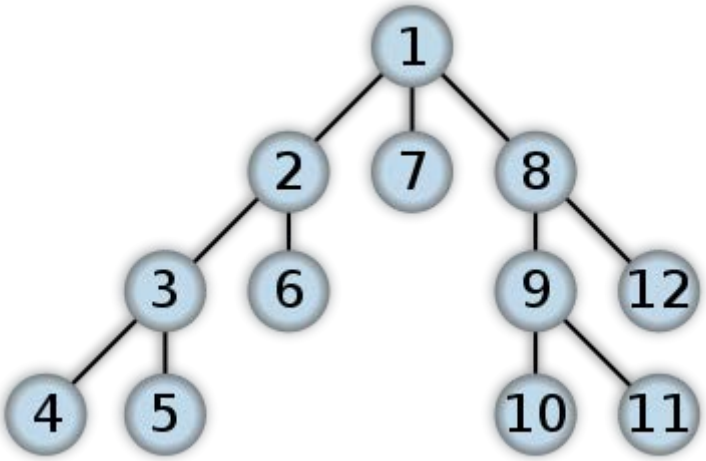
7.5. Створіть програму, що виводить всі ребра графу у порядку зростання ваги.

7.6. Використовуючи бібліотеку tkinter, створіть програму для візуального зображення графу.

§7.2. Алгоритми на графах

Пошук в глибину

Алгоритм пошуку в глибину (англ. Depth-first search, DFS) — алгоритм для обходу графа у якому застосовується стратегія йти, на скільки це можливо, вглиб графа,. Прикладом пошуку в глибину може бути стратегія пошуку потрібного файлу у ієрархії папок. Нижче наведено порядок обходу графа починаючи з вершини на якій зазначено 1.



Алгоритм

Як правило DFS реалізується рекурсивно. Ідея рекурсивного алгоритму пошуку в глибину полягає у тому, що починаючи з обраної вершини в графі, потрібно здійснити пошук в глибину для всіх її сусідів.

Запишемо алгоритм пошуку в глибину, використовуючи псевдокод:

```
def DFS(start):
    Опрацювати початкову вершину start
    Помітити вершину start як відвідану
    for всіх сусідів neighbour вершини start які ще не були відвідані :
        DFS(neighbour)  # запускаємо DFS
```

Реалізація

Реалізуємо пошук в глибину на Python використовуючи клас `Graph` реалізований вище на базі списку суміжності. Для автономності рекурсивної підпрограми будемо передавати у підпрограму у ролі параметрів не лише стартову точку, але і сам граф, а також список відвіданих вершин. Тоді підпрограма буде мати вигляд

```
def DFS(graph, visited, start):
    print(start, end = " ")           # Опрацьовуємо стартовий елемент
    visited[start] = True  # Помічаємо стартовий елемент як відвіданий
    # для всіх сусідів поточного елементу
    for neighbour in graph[start].getNeighbors():
        if not visited[neighbour]:  # які ще не були відвідані
            DFS(graph, visited, neighbour)  # запускаємо DFS
```

Щоб перевірити роботу алгоритму створимо граф зображений на рисунку 2 та запустимо для нього пошук в глибину. Пошук в глибину у нашому випадку виведе на екран послідовність обходу вершин у графі

```
g = Graph(True) # Створюємо орієнтований граф

g.addEdge(0, 1)
g.addEdge(0, 5)
g.addEdge(1, 2)
g.addEdge(2, 3)
g.addEdge(3, 4)
g.addEdge(3, 5)
g.addEdge(4, 0)
g.addEdge(5, 4)
g.addEdge(5, 2)
```

```
# Створюємо список відвіданих вершин, які
# ініціалізуються значенням False
visited = [False] * len(g)

DFS(g, visited, 0) # запускаємо DFS з вершини 0
```

Результатом роботи буде виведення на екран

```
0 1 2 3 4 5
```

Зауваження. Наведений вище алгоритм буде працювати доки не будуть пройдені всі вершини у графі, що є досяжними з стартової точки. Його важко модифікувати (а якщо можна, то це не є ефективним), так, щоб алгоритм припиняв свою роботу у випадку відшукання певної вершини, що є метою пошуку, оскільки, принаймні, необхідно вийти з усіх рекурсивних викликів функції DFS. Тому використання алгоритму пошуку в глибину виправданим, якщо у будь-якому разі необхідно пройти всі вершини, що досяжні з заданої.

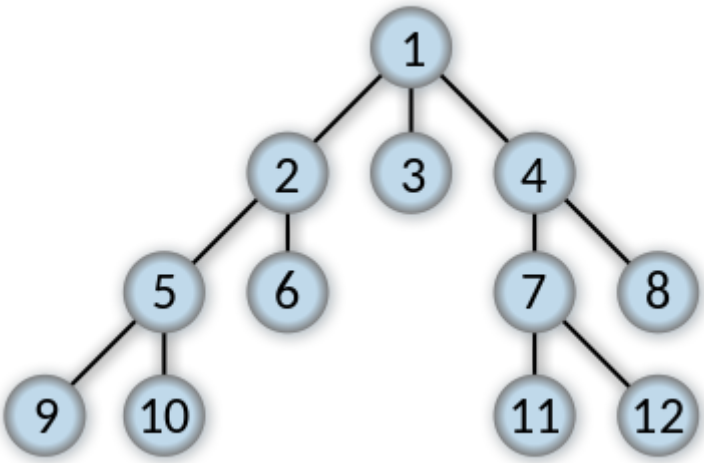
Отже, алгоритм пошуку в глибину використовується для розв'язання задач, що пов'язані з досяжністю однієї вершини з іншої, причому довжина шляху не є важливою. Таким чином, DFS використовується для знаходження циклів у графі, перевірки чи є граф зв'язним тощо.

Алгоритмічна складність алгоритму пошуку в глибину

Алгоритмічна складність пошуку в глибину залежить від того, як реалізований граф. Так, наприклад, у випадку реалізації графа через матрицю суміжності алгоритмічна складність пошуку в глибину буде $O(n^2)$ якщо граф має n вершин. Тоді, як для у випадку реалізації графу через список суміжності, його алгоритмічна складність буде порядку m у середньому, та $O(n+m)$ у найгіршому випадку, де m – кількість ребер у графі.

Пошук в ширину

Пошук у ширину (англ. breadth-first search, BFS) — алгоритм пошуку на графі, у якому застосовується стратегія послідовного перегляду окремих рівнів графа, починаючи з заданого вузла. Нижче наведено порядок обходу графа починаючи з вершини на якій зазначено 1.



Алгоритм

Для послідовного перегляду окремих рівнів графа, починаючи з заданого вузла, використовується черга. Спочатку у цю чергу додається стартова вершина графа, яка зразу позначається як така, що відвідана. Далі алгоритм полягає у такому, що поки ця черга не порожня ми вилучаємо з черги черговий елемент, опрацьовуємо його, додаємо до черги всіх його сусідів, що не були до цього відвідані, позначаючи їх відвіданими.

Формально алгоритм пошуку в ширину можна записати, використовуючи псевдокод, таким чином:

```
def BFS(start):
    Додати в чергу початкову вершину start
    Помітити вершину start як відвідану

    while черга не порожня:
```

```
        Взяти з черги елемент current
        Опрацювати вершину current
        for всіх сусідів neighbour вершини current які ще не були відвідані :
            Додати в чергу neighbour
            Помітити neighbour як відвідану
```

Зауважимо, що наведений вище алгоритм також виконується доки не будуть пройдені всі вершини графа, досяжні зі стартової. Проте, цей алгоритм легко модифікувати, якщо задача поставлена знайти потрібний елемент у графі. Для цього, якщо цільова вершина знайдена достатньо всього лиш перервати роботу циклів.

Реалізація

Реалізуємо пошук в ширину на Python використовуючи клас `Graph` реалізований вище на базі списку суміжності.

```python
def BFS(graph, start):

    # Введемо масив, що буде містити ознаку чи відвідали вже вершину.
    # Ініціалізуємо масив значеннями False (тобто відвідали)
    visited = [False] * len(graph)

    q = Queue() # Створюємо чергу

    q.pushBack(start)      # Додаємо у чергу стартову вершину
    visited[start] = True  # та позначаємо її як відвідану

    while not q.empty():   # Поки черга не порожня

        current = q.popFront()  # Беремо перший елемент з черги
        print(current)          # Опрацьовуємо взятий елемент

        # Додаємо в чергу всіх сусідів поточного елементу
        for neighbour in graph[current].getNeighbors():
            if not visited[neighbour]:  # які ще не були відвідані
                q.pushBack(neighbour)
                visited[neighbour] = True  # Помічаємо neighbour як відвідану
```

Алгоритмічна складність

Як і у випадку пошуку в глибину, алгоритмічна складність пошуку в ширину залежить від того, як реалізований граф. У випадку реалізації графа через матрицю суміжності алгоритмічна складність пошуку в ширину буде $O(n^2)$ якщо граф має n вершин. Тоді, як для у випадку реалізації графу через список суміжності, його алгоритмічна складність буде $O(n+m)$ у найгіршому випадку, де m – кількість ребер у графі.

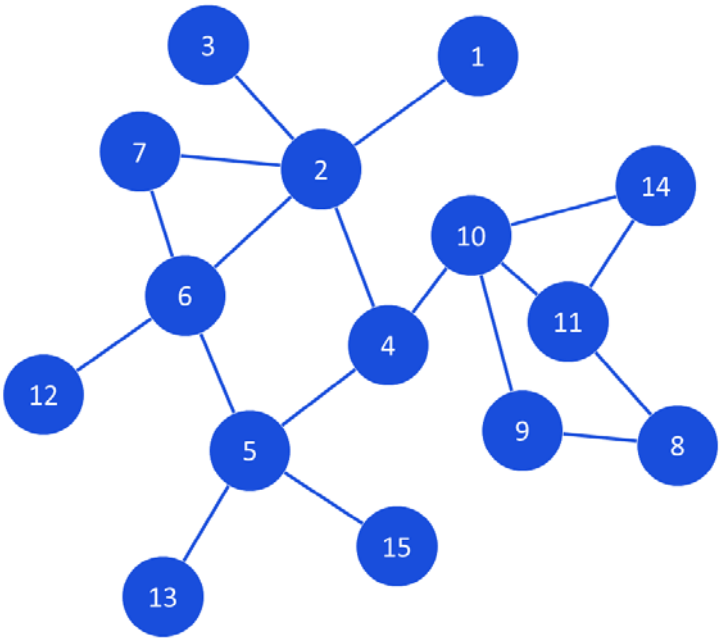
Хвильовий алгоритм

Алгоритм

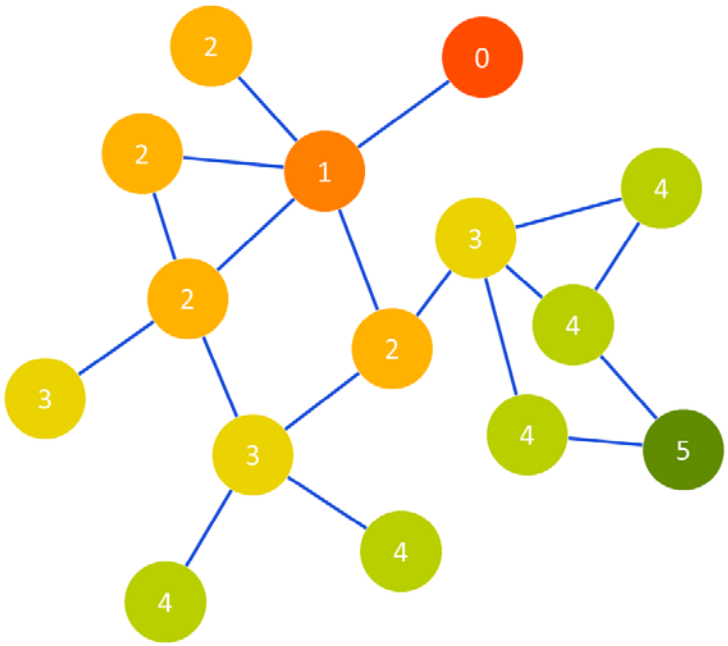
Виявляється пошук в ширину не лише може знайти вершину у графі, якщо вона досяжна з деякої заданої, але і знайти довжину найкоротшого шляху до цієї вершини зі стартової. Найпростіший спосіб здійснити це – застосувати хвильовий алгоритм, який базується на пошуку в ширину.

Ідея хвильового алгоритму полягає у тому, що ми запускаємо BFS і при цьому для кожної вершини, крім того чи відвідана вона чи ні, ми ще будемо пам'ятати відстань від стартової точки до неї. Для цього будемо вважати, що відстань від стартової точки до себе є нулем. А далі під час додавання точки до черги (якщо вона ще не була відвідана) будемо встановлювати для неї цю відстань на одиницю більшу ніж для поточної.

Наприклад, для такого нерієнтованого графа



Відстань від вершини 1 буде схематично зображувати таким чином



Звідки можна зробити висновок, що, наприклад, найкоротша відстань від точки 1 до точки 8 буде дорівнювати 5.

Хвильовий алгоритм можна формально записати так:

```
def wave(start):
    Додати в чергу початкову вершину start
    Помітити вершину start як відвідану
    Встановити відстань від стартової точки start до себе нуль

    while черга не порожня:
        Взяти з черги елемент current
        Опрацювати вершину current
        for всіх сусідів neighbour вершини current які ще не були відвідані :
            Додати в чергу neighbour
            Помітити neighbour як відвідану
            Встановити відстань для neighbour на одиницю більшу ніж для поточної
```

Реалізація

Наведемо реалізацію хвильового алгоритму.

```
def wave(graph, start):

    # Введемо масив, що буде містити ознаку чи відвідали вже вершину.
    # Ініціалізуємо масив значеннями False (тобто відвідали)
    visited = [False] * len(graph)

    # Введемо масив, що буде містити
    # відстані від стартової вершини start.
    # Ініціалізуємо масив значеннями -1 (тобто нескінченність)
    distances = [-1] * len(graph)

    q = Queue() # Створюємо чергу

    q.pushBack(start)       # Додаємо у чергу стартову вершину
    visited[start] = True   # та позначаємо її як відвідану
    distances[start] = 0    # Відстань від стартової точки до себе нуль.

    while not q.empty():

        current = q.popFront()  # Беремо перший елемент з черги

        # Тут Опрацьовуємо взятий елемент, за необхідності

        # Додаємо в чергу всіх сусідів поточного елементу
        for neighbour in graph[current].getNeighbors():
            if not visited[neighbour]:  # які ще не були відвідані
                q.pushBack(neighbour)
                visited[neighbour] = True
                # Встановлюємо відстань на одиницю більшу ніж для поточної
                distances[neighbour] = distances[current] + 1

    # Повертаємо масив відстаней від start до всіх точкок графа
    return distances
```

Зауважимо, що вищенаведений алгоритм можна дещо оптимізувати по використанню пам'яті. Дійсно, список `visited` відвіданих вершин нам не потрібний, оскільки вияснити питання чи відвідана вершина ми можемо використовуючи масив `distances` відстаней від стартової. Отже, якщо вершина не була відвідана, то у відповідному полі масиву `distances` стоїть -1 (так ми в алгоритмі позначили нескінченність). Якщо ж вершина відвідана, там записана довжина найкоротшого шляху від стартової вершини. Пропонуємо провести таку оптимізацію читачу самостійно.

Операції з графами

Основні операції для незваженого графу, вершини якого не мають додаткового навантаження такі

1. Створення нового порожнього графу.
2. Операція додавання в граф нової вершини.
3. Операція додавання в граф нового ребра, що з'єднує дві вершини (для неорієнтованого графу).
4. Операція додавання в граф нової дуги, що з'єднує дві вершини (для неорієнтованого графу).
5. Пошук у графі вершини.
6. Отримання списку всіх вершин графа.
7. Перевірка чи входить задана вершина в граф.

Для зважених графів, або графів з додатковим навантаженням на вершини потрібно додати відповідні операції

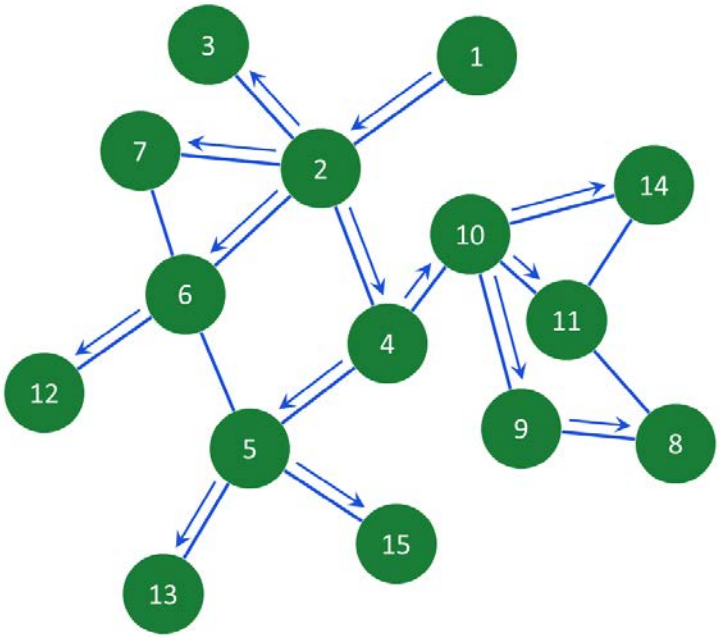
1. Отримати/встановити навантаження вершини.
2. Отримати/встановити вагу ребра

Пошук найкоротшого шляху

Алгоритм

Вище наведено алгоритм як знайти найкоротшу відстань між двома вершинами у графі. Проте питання про те як знайти сам цей найкоротший шлях поки що залишалося відкритим. Ідея такої модифікації алгоритму, щоб знайти найкоротший шлях полягає у тому, що нам потрібно не лише

запам'ятовувати відстань від стартової точки до заданої, але ще і вершину (тобто маркер) з якої ми прийшли по цьому найкоротшому шляху. Тоді, для відшукання шляху необхідно буде лише відновити шлях від кінцевої точки до початкової по встановлених маркерах. Для прикладу, розглянемо вищенаведений граф. Запам'ятаємо вершини з яких ми прийшли, як показано на рисунку нижче.



Тепер, як бачимо, наприклад, найкоротший шлях від точки 1 до точки 8 можна знайти як:

- у 8 ми прийшли з 9
- у 9 ми прийшли з 10
- у 10 ми прийшли з 4
- у 4 ми прийшли з 2
- у 2 ми прийшли з 1

Таким чином шлях буде виглядати $1 \to 2 \to 4 \to 10 \to 9 \to 8$.

Як бачимо для того щоб побудувати шлях, нам треба пройти за маркерами у зворотному порядку. Таку операцію зручно робити за допомогою стеку.

Реалізація алгоритму

```
def findingWay(graph, start, end):
    assert start != end

    distances = [-1] * len(graph)    # Масив відстаней
    sources = [-1] * len(graph)      # Масив вершин звідки прийшли

    q = Queue()             # Створюємо чергу

    q.pushBack(start)       # Додаємо у чергу стартову вершину
    distances[start] = 0    # Відстань від стартової точки до себе нуль.

    while not q.empty():

        current = q.popFront()  # Беремо перший елемент з черги

        # Додаємо в чергу всіх сусідів поточного елементу
        for neighbour in graph[current].getNeighbors():
            if distances[neighbour] == -1:  # які ще не були відвідані
                q.pushBack(neighbour)
                distances[neighbour] = distances[current] + 1
                sources[neighbour] = current # Вказуємо для сусіда neighbour,
                                             # що ми прийшли з вершини current

    # будуємо шлях за допомогою стеку
    stack = Stack()
```

```
    current = end
    while True:
        stack.push(current)
        if current == start:
            break
        current = sources[current]

    # виводимо шлях на екран
    while not stack.empty():
        v = stack.pop()
        if not stack.empty():
            print(v, end=" -> ")
        else:
            print(v)
```

Топологічне сортування

§7.3. Алгоритми на зважених графах

Алгоритм Беллмана-Форда

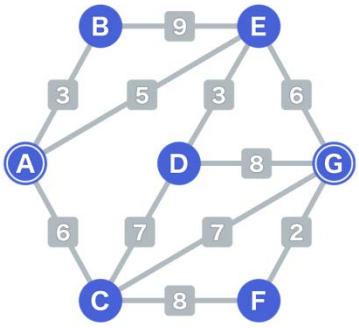
```
def BelmanFordClasical(graph, start):
    """ Класичний алгоритм Беллмана-Форда для графів,
        що можуть мати цикли від'ємної ваги
    """

    # Ініціалізуємо додаткову інформацію у графі для роботи алгоритму
    # Відстань для кожної вершини від стартової ставиться як нескінченність
    for vert in graph:
        vert.reinit()

    # Відстань від першої вершини до неї ж визначається як 0
    graph[start].setDistance(0)
    for i in range(len(graph) - 1):
        for current in graph:                                # Для всіх вершин графу
            for neighbourId in current.getNeighbors():       # Для сусідів поточного елементу
                neighbour = graph[neighbourId]               # Беремо сусіда за індексом
                neighbourDist = neighbour.getDistance()  # Беремо поточне значення
відстані у вершині-сусіді
                newDist = current.getDistance() + current.getWeight(neighbourId)    #
Обчислюємо потенційну відстань у вершині-сусіді
                if newDist < neighbourDist:                  # Якщо потенційна відстань у
вершині-сусіді менша за її поточне значення
                    neighbour.setDistance(newDist)           # Змінюємо поточне значення
відстані у вершині-сусіді обчисленим
                    neighbour.setFrom(current.getId())   # Встановлюємо для сусідньої
вершини ідентифікатор звідки ми прийшли у неї
```

Алгоритм Дейкстри

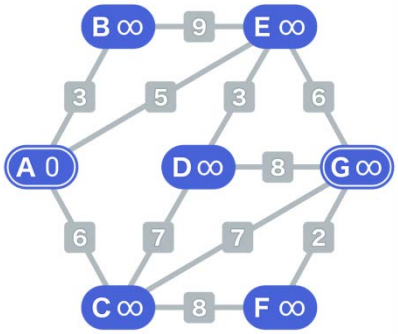
У випадку, якщо граф зважений, то для пошуку найкоротшого шляху використовувати звичайний алгоритм BFS вже не можливо. Одним з алгоритмів, які дозволяють розв'язати таку задачу у зважених графах, що не мають циклів та ребер від'ємної ваги є алгоритм Дейкстри названий на честь нідерландського математика Едсгера Дейкстри (нід. Edsger Wybe Dijkstra), який його власне і відкрив. За допомогою цього алгоритму можна знайти найкоротший шлях від однієї фіксованої вершини графа до всіх інших його вершин.

Алгоритм

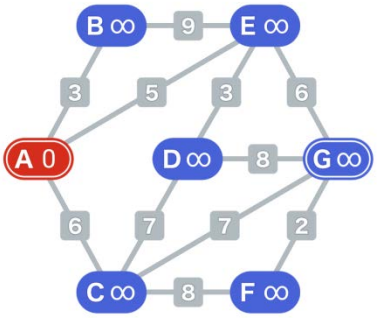
Для пояснення алгоритму розглянемо граф зображений на рисунку нижче. Будемо шукати найкоротшу відстань від вершини A до вершини G.



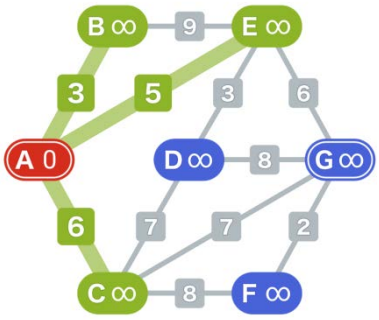
У алгоритмі Дейкстри використовується додаткове навантаження на вершини графу: ми визначаємо загальну вагу, тобто відстань, найкоротшого шляху від стартової вершини до кожної вершини. Алгоритм Дейкстри є ітеративним. На першій ітерації вважається, що відстань від стартової точки A до себе є нульовою, а відстані до всіх інших точок графу є нескінченними. На наступному рисунку поруч з ідентифікатором вершини зазначається навантаження вершин на першій ітерації.



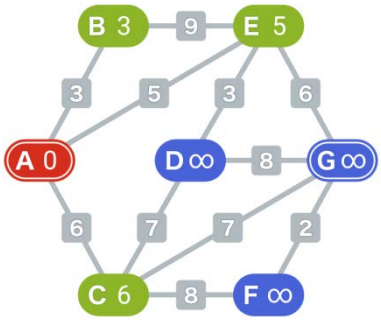
Далі відбувається пошук із початкової точки A. При цьому поточна вершина A стає фіксованою і її навантаження уже в подальшому не буде змінюватися.



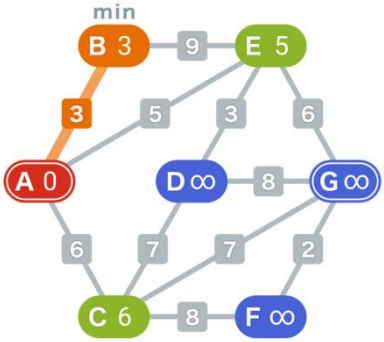
На кожній ітерації визначається наступна вершина графу, до якої можна дістатися зі стартової вершини найкоротшим шляхом (у рахуванням уже пройденої відстані до поточної вершини) серед точок, що є сусідами поточної. Як бачимо, у вершини A є три сусіди: B, C, E.

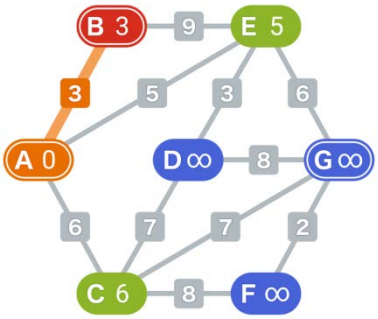


Модифікуємо навантаження для кожної з цих вершин, як суму навантаження точки A та ваг відповідних ребер, якщо ця сума менша за поточне навантаження відповідних вершин.

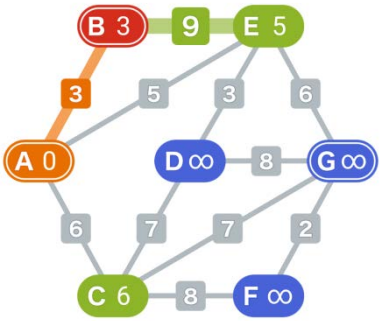


Далі, серед усіх вершин, за виключенням фіксованих (на цій ітерації це лише вершина A), знаходимо вершину, що має найменше навантаження. Очевидно, що це буде вершина B. Тепер вершина B стає фіксованою, і подальший пошук відбувається уже саме з цієї вершини.

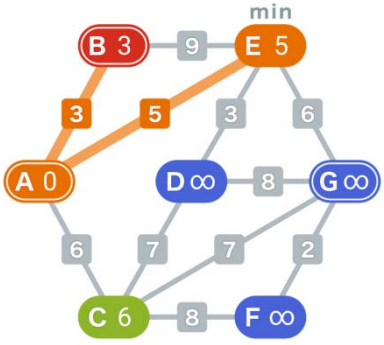




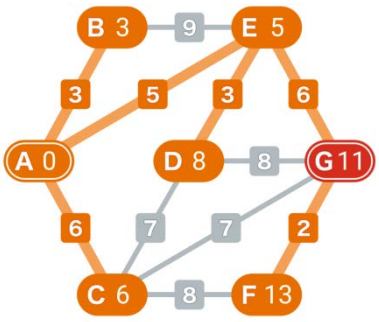
Далі з вершини B можна потрапити (за виключенням тих, які вже фіксовані) лише у вершину E.



Оскільки сума навантаження вершини B (3) та вага ребра BE (9), яка дорівнює 12 є більшою за поточне навантаження вершини E (5), то лишаємо навантаження вершини E без змін і знову знаходимо серед всіх незафіксованих вершин, вершину з найменшим навантаженням. На цій ітерації такою буде вершина E, яка у цей момент стає фіксованою, і подальший пошук проводиться вже з цієї точки.



Цей процес проводиться доти, доки всі точки графу не стануть фіксованими.



Отже, як бачимо, ми знайшли не лише довжину найкоротшого шляху від вершини A до вершини G, але й найкоротші відстані до всіх вершин графу. Наприклад, відстань до вершини F є 13. При цьому, якщо, запам'ятовувати вершину з якої прийшли у поточну, то легко відтворити й сам шлях.

Наведемо описаний процес роботи алгоритму для зазначеного графа у вигляді таблиці. Поточну точку на кожній ітерації будемо виділяти за допомогою фону іншого кольору у відповідній клітині таблиці.

A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞
	3	6	∞	5	∞	∞
		6	∞	5	∞	∞
		6	8		∞	11
			8		14	11
					14	11
					13	

Тепер можемо формально записати алгоритм Дейкстри за допомогою псевдокоду.

```
def Dijkstra(start):
    Встановити навантення для вершини start як нуль
    Встановити навантення для всіх інших вершин графа як нескінченність

    Додати в пріоритетну чергу всі вершини графа

    while пріоритетна черга не порожня:
        Взяти з черги елемент current з найвищим пріоритетом
        Опрацювати вершину current
        for всіх сусідів neighbour вершини current:
            обчислюємо dist як суму навантаження current
                    та ваги ребера з current до neighbour
            if dist < навантаження клітини current:
                модифікуємо навантаження neighbour значенням dist
```

Реалізація

Для реалізації, розширимо клас `GraphVertex`, що використовувався для задавання вершин графа, так, щоб ми пам'ятали у вершині поточну найменшу відстань від стартової вершини, а також з якої вершини ми прийшли.

```
class VertexDijkstra(GraphVertex):
    def __init__(self, aKey):
        super().__init__(aKey)

        self.mDistance = 100000  # Вказуємо величину найкоротшого шляху до даної
                                 # вершини графа від деякої іншої фіксованої
                                 # вершини. Спочатку вона є несінченністю!
        self.mFrom = None        # Вказує звідки ми прийшли по найкорошому шляху

    def setDistance(self, aDistance):
        self.mDistance = aDistance
```

```
    def getDistance(self):
        return self.mDistance

    def setFrom(self, aFrom):
        self.mFrom = aFrom

    def getFrom(self):
        return self.mFrom
```

Зауважимо, що цей клас також зручно було б використовувати для хвильового алгоритму та алгоритму пошуку найкоротшого шляху наведених вище. Пропонуємо читачеві самостійно модифікувати вищенаведені алгоритми з використанням класу `VertexDijkstra`.

Алгоритм Дейкстри використовує структуру даних пріоритетну чергу для визначення вершини, що має найменше навантаження на поточному кроці.

Нарешті, алгоритм Дейкстри буде мати вигляд

```
def Dijkstra(graph, start, end):
    q = PriorityQueue()           # Створюємо пріоритетну чергу
    graph[start].setDistance(0)

    # Ініціалізуємо чергу з пріоритетом, де пріоритет це відстань вершини
    # Спочатку всі відстані крім першої вершини ставляться як нескінченність
    for v in graph:
        d = v.getDistance()
        q.insert(d, v)

    while not q.empty():
        current = q.extract_minimum() # Беремо елемент з черги
        for neighbour in current.getNeighbors():
            newDist = current.getDistance() + current.getWeight(neighbour)
            neighbourDist = graph[neighbour].getDistance()
            if newDist < neighbourDist:
                graph[neighbour].setDistance(newDist)
                # Після зміни відстані у вершині-сусіді потрібно обов'язково
                # перерахувати пріоритети в черзі. Наша реалізація пріоритетної
                # черги проводить перерахунок приоритетів під час витягування
                # елемента з черги, тому тут це робити зараз не треба
                graph[neighbour].setFrom(current.getId())  # Встановлюємо для
                    # сусідньої вершини ідентифікатор звідки ми прийшли у неї

    # будуємо шлях за допомогою стеку
    stack = Stack()
    current = end
    while True:
        stack.push(current)
        if current == start:
            break
        current = graph[current].getFrom()

    # виводимо шлях на екран
    while not stack.empty():
        v = stack.pop()
        if not stack.empty():
            print(v, end=" -> ")
        else:
            print(v)
```

A* алгоритм

Завдання для самостійної роботи

Задачі з сайту e-olimp.com:
Пошук в ширину та глибину, хвильовий алгоритм, пошук найкоротшого шляху

2401, 4852, 4853, 982, 4007
Алгоритм Дейкстри: 1365, 1388

§7.4. Пошуки шляхів у лабіринтах

Теорія графів дозволяє розв'язувати найрізноманітніші задачі, серед яких алгоритми обробки лабіринтів. Окремим класом лабіринтів, є лабіринти, що зображуються двомірною (або тривимірною) прямокутною сіткою у якій кожна клітина має певне (наперед визначене) навантаження – тобто її стан (вільна клітина, стіна, тощо). Причому, вважається, що клітина сітки одночасно може бути лише у одному з кількох визначених наперед станів.

Як правило, всі лабіринти у таких задачах мають клітини, що є

- вільними – клітини лабіринту у які можна пересуватися з інших клітин;
- зайнятими – клітини лабіринту, пересування у які забороняється умовою задачі (стіна, небезпечна ділянка).

На рисунку 6 зображено двовимірний лабіринт у якому білим кольором позначаються вільні клітини, а сірим – відповідно зайняті.

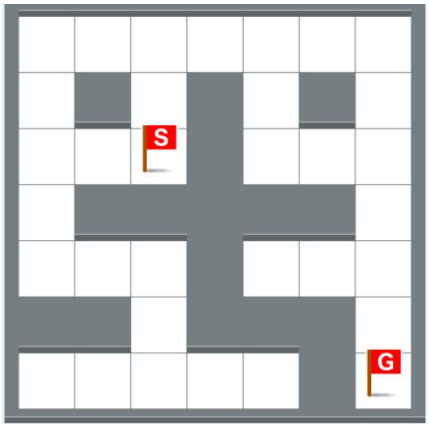


Рисунок 6

Крім зазначених вище типів клітин, можуть бути й інші, навантаження яких додатково визначається умовою задачі.

Рух у такому лабіринті за один крок допускається лише з поточної до сусідніх клітин. При цьому рух з поточної до сусідніх клітин, для двовимірного (тобто, плоского лабіринту) може здійснюватися у чотирьох або восьми напрямках (що визначається умовою задачі)

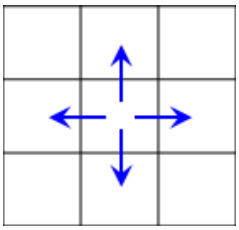


Рисунок 7

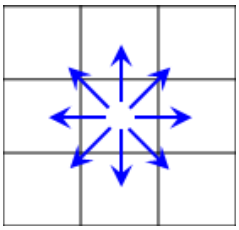


Рисунок 8

У тривимірному лабіринті напрямків буде відповідно 6 або 26. Пропонуємо читачеві самостійно схематично їх зобразити.

Моделювання лабіринту

Зображення лабіринту за допомогою матриці

Зображення лабіринту у пам'яті комп'ютера звичайно можна провести за допомогою графів (тобто матриці або списку суміжності) як це здійснювалося вище. Проте це не є зручним способом, як з точки зору зручності для подальшого програмування, так і з точки зору швидкодії. Тому у таких задачах доцільно для зображення лабіринту використовувати двовимірні або тривимірні матриці, елементами яких буде навантаження клітини сітки лабіринту. Наприклад, для вищенаведеного на рисунку 6 лабіринту використаємо таку матрицю

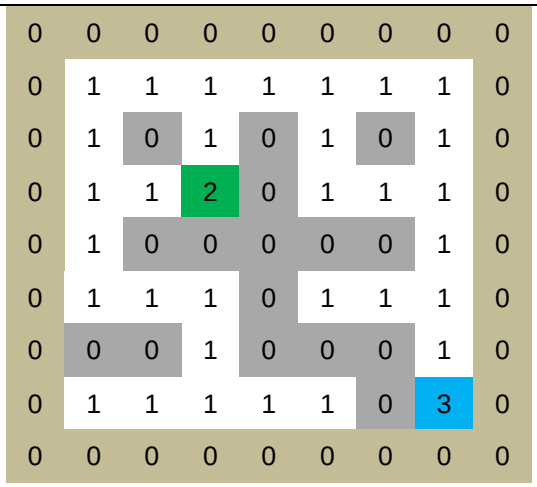


Рисунок 9. Матриця, що визначає конфігурацію лабіринту.

у якій 0 буде позначати стіну, тобто зайняту клітину, 1 – вільну клітину у яку можливий перехід, 2 – початкову позицію у лабіринті з якої починається рух по лабіринту та 3 – кінцеву точку шляху.

Звернемо увагу читача, що хоча лабіринт на рисунку 6 має розмірність 7x7, ми використали для зображення лабіринту матрицю 9x9 у якій два додаткових рядки та стовпчики, що заповнені 0 використовуються для того, щоб позначити зовнішні границі лабіринту. Такий підхід не є обов'язковим, проте у подальшому дозволяє спростити написання алгоритму, оскільки зникає необхідність у перевірці умов виходу за межі масиву (тобто за межі лабіринту).

Таким чином, для створення лабіринту, що має

```
N = 7 # рядків сітки лабіринту
M = 7 # стовпчиків сітки лабіринту
```

та спочатку має лише зайняті клітини можна використати такий код

```
maze = []  #  Створення порожньої матриці для задавання лабіринту
for i in range(N + 2):
    row = [0] * (M + 2)  # рядок матриці з M+2 елементів, що складається з 0
    maze.append(row)
```

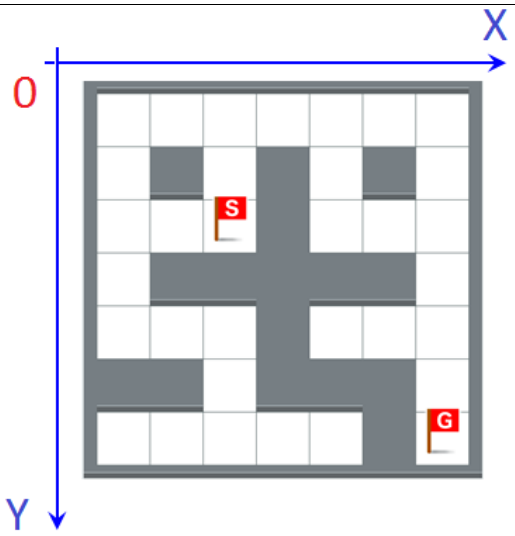
Далі необхідно зазначити всі вільні клітини, наприклад, явно вказуючи їх у програмі

```
maze[1][1] = 1 # вільна клітина лабіринту
maze[1][2] = 1 # вільна клітина лабіринту
......
maze[3][3] = 2 # стартова клітина лабіринту
maze[7][7] = 3 # кінцева клітина лабіринту
```

Функція виведення лабіринту на екран, що неодмінно знадобиться принаймні для тестування роботи алгоритму, буде мати вигляд

```
def showMaze(maze):
    for row in maze:
        print(*row)
```

Зауваження. Тут варто привернути увагу читача, до того факту, що лабіринт задається матрицею, у якій нумерація рядків йде згори до низу. Фактично, з точки зору координатного підходу це означає, що початок системи координат лабіринту знаходиться у лівому верхньому куті лабіринту, а вісь 0Y направлена донизу.



Крім цього зазначимо, що оскільки лабіринт задається матрицею `maze`, доступ до елементів у якій за домовленістю здійснюється із зазначенням номера рядка та стовпчика

```
maze[1][2] # клітина лабіринту, що знаходиться у 1-му рядку та 2-му стовпчику
```

то фактично з точки зору координатного підходу, першою задається координата `y`, а вже другою `x`:

```
maze[y][x] # клітина лабіринту, що знаходиться у y-му рядку та x-му стовпчику
```

Зчитування лабіринту з клавіатури

Очевидно, що можна використовувати інший спосіб задавання типу клітин лабіринту у матриці. Наприклад, якщо матриця лабіринту зчитується з клавіатури по рядках, то можна використати такий спосіб

```
def inputMaze(N, M):
    maze = []  # Створення порожньої матриці для задавання лабіринту

    row0 = [0] * (M + 2)  # перший рядок матриці, що визначає верхню стіну
    maze.append(row0)

    for i in range(N):
        str_row = input() # Зчитування рядка з клавіатури

        # Перетворення рядка у список цілих чисел
        row = list(map(int, str_row.split()))

        # додавання лівої та правої "стіни" лабіринту
        row.insert(0, 0)
        row.append(0)

        maze.append(row) # додавання рядка до лабіринту

    rowLast = [0] * (M + 2)  # останній рядок матриці, що визначає нижню стіну
    maze.append(rowLast)

    return maze # Повертаємо створений лабіринт
```

Тут зчитування лабіринту оформлено у вигляді підпрограми, що вхідними параметрами має кількість рядків та стовпчиків у лабіринті, відповідно. Виклик цієї функції виглядатиме так

```
maze = inputMaze(7, 7)
```

Зчитування лабіринту з текстового файлу

Якщо ж матриця лабіринту зчитується з файлу по рядках (і саме такий спосіб рекомендується використовувати для тестування ваших власних програм), то можна використати код наведений

нижче. Як і попередній випадок цей код оформлений у вигляді підпрограми, яка першим параметром має ім'я файлу у якому зберігається лабіринт.

```
def readMazeFromFile(aFileName, N, M):
    maze = []  # Створення порожньої матриці для задавання лабіринту

    row0 = [0] * (M + 2)  # перший рядок матриці, що визначає верхню стіну
    maze.append(row0)

    # Зчитування лабіринту з файлу
    with open(aFileName) as f:
        for str_row in f:
            # Перетворення рядка у список цілих чисел
            row = list(map(int, str_row.split()))

            if len(row) == 0: # Захист від зайвих рядків у кінці файлу
                break

            # додавання лівої та правої "стіни" лабіринту
            row.insert(0, 0)
            row.append(0)

            maze.append(row) # додавання рядка до лабіринту

    rowLast = [0] * (M + 2)  # останній рядок матриці, що визначає нижню стіну
    maze.append(rowLast)

    return maze # Повертаємо створений лабіринт
```

Виклик цієї підпрограми матиме вигляд

```
maze = readMazeFromFile("maze1.txt", 7, 7)
```

де, файл `maze1.txt` міститься у тій же папці, що і файл програми та має такий вміст

```
1 1 1 1 1 1 1
1 0 1 0 1 0 1
1 1 2 0 1 1 1
1 0 0 0 0 0 1
1 1 1 0 1 1 1
0 0 1 0 0 0 1
1 1 1 1 1 0 3
```

Рух по лабіринту

Для пошуку у лабіринтах використовуються ті ж стандартні алгоритми обходу графів, такі як пошук DFS, BFS, хвильовий алгоритм, алгоритм Дейкстри, тощо. Розглянемо модифікації цих алгоритмів, для двомірних лабіринтів.

Для зручного (з точки зору програмування) пересування по лабіринту з деякої поточної клітини до сусідніх використовується два допоміжних списки

```
dx = [-1,  0, 1, 0]
dy = [ 0, -1, 0, 1]
```

у випадку якщо рух можливий лише по вертикалі та горизонталі, тобто у напрямках зазначених на рисунку 7, та списки

```
dx = [-1, -1, 0, 1, 1, 1, 0, -1]
dy = [0, -1, -1, -1, 0, 1, 1, 1]
```

якщо допускається рух по діагоналі (див рисунок 8).

Тоді, щоб відвідати всіх сусідів клітини з координатами `(i,j)` потрібно виконати такий код

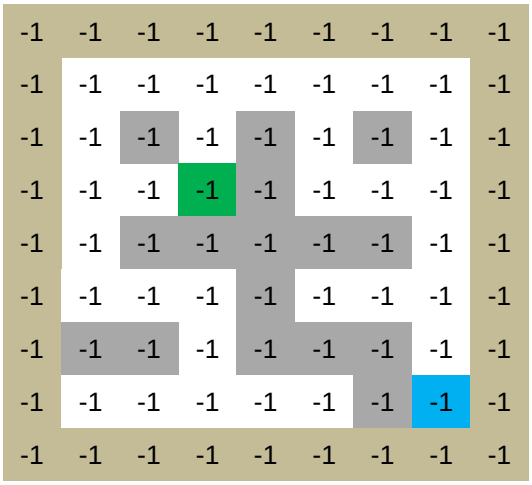
```
for k in range (len(dx)):
    process(i+dy[k], j+dx[k])
```

де `process(i,j)` деякий код, що виконується над клітиною лабіринту з координатами `(i,j)`.

Пошук в глибину та хвильовий алгоритм

Пошук в глибину чи хвильовий алгоритм для лабіринтів майже нічим не відрізняється від відповідних стандартних алгоритмів для графів. Проте алгоритм пошуку власне найкоротшого шляху має певну особливість, яка дозволяє значно економити оперативну пам'ять комп'ютера. Отже розглянемо спочатку хвильовий алгоритм.

Для його застосування використовується допоміжна матриця тієї ж розмірності, що і матриця лабіринту, призначення якої зберігати інформацію про відвідані клітини лабіринту (у випадку простого BFS цього досить), а також відстань від стартової точки лабіринту до поточної. Як правило така матриця ініціалізується значенням -1 для всіх клітин. Цю допоміжну матрицю будемо називати хвильовою матрицею лабіринту.



Зауважимо, що при цьому немає значення яким чином ініціалізуються клітини хвильової матриці, що відповідають зайнятим клітинам лабіринту.

Для стартової точки, з якої починається пошук у лабіринті, очевидно, у хвильовій матриці встановлюється значення 0. Далі алгоритм повністю повторює наведений вище хвильовий алгоритм, за виключенням того, що у чергу додаються не вершини графу, а координати сусідніх клітин які ще до цього не були відвідані.

Нижче наведено підпрограму, що реалізує хвильовий алгоритм. Вхідними параметрами будуть матриця лабіринту `maze` та початкова клітина пошуку `start`. Після виконання, підпрограма поверне хвильову матрицю.

```
def wave(maze, start):
    # maze - матриця лабіринту
    # start - початкова позиція у лабіринті у вигляді кортежу (рядок, стовпчик)

    dx = [-1, 0, 1, 0]
    dy = [0, -1, 0, 1]

    n = len(maze)    # кількість рядків у матриці maze
    m = len(maze[0]) # кількість стовпчиків у матриці maze

    # створення та ініціалізація хвильової матриці
    # такої ж розмірності, що і матриця лабіринту
    waveMatrix = []
    for i in range(n):
        row = [-1] * m
        waveMatrix.append(row)

    q = Queue()         # Створюємо чергу
    q.pushBack(start)  # Додаємо у чергу координати стартової клітини
    waveMatrix[start[0]][start[1]] = 0  # Відстань від стартової до себе нуль

    while not q.empty():

        current = q.popFront()  # Беремо перший елемент з черги
```

```
        i = current[0]  # координата поточного рядка матриці
        j = current[1]  # координата поточного стовчика матриці

        # Додаємо в чергу всі сусідні клітини
        for k in range (len(dx)):

            i1 = i+dy[k]  # координата рядка сусідньої клітини
            j1 = j+dx[k]  # координата стовпчика сусідньої клітини

            # які ще не були відвідані та у які можна пересуватися
            if waveMatrix[i1][j1] == -1 and maze[i1][j1] != 0:
                q.pushBack((i1, j1))
                # Встановлюємо відстань на одиницю більшу ніж для поточної
                waveMatrix[i1][j1] = waveMatrix[i][j] + 1

    # Повертаємо хвильову матрицю
    return waveMatrix
```

Виклик цієї підпрограми для вищезазначеного лабіринту `maze` та початкової клітини `(3,3)` буде таким

```
waveMatr = wave(maze, (3, 3))
```

результатом виконання якої буде хвильова матриця `waveMatr` зображена нижче

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	4	3	2	3	4	5	6	-1
-1	3	-1	1	-1	5	-1	7	-1
-1	2	1	0	-1	6	7	8	-1
-1	3	-1	-1	-1	-1	-1	9	-1
-1	4	5	6	-1	12	11	10	-1
-1	-1	-1	7	-1	-1	-1	11	-1
-1	10	9	8	9	10	-1	12	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

Як бачимо, по цій матриці легко встановити відстань від клітини `(3,3)` до будь-якої клітини лабіринту, зокрема, відстань до клітини `(7,7)` буде дорівнювати 12.

Відшукання шляху

Для відшукання шляху, під час виконання хвильового алгоритму, як і для графі можна було запам'ятовувати для кожної клітини, координати клітини з якої ми прийшли. Проте, ця процедура виявляється надлишковою, оскільки маючи хвильову матрицю можна легко відновити шлях. Дійсно, для цього достатньо рухатися, починаючи з кінцевої точки у лабіринті, від сусіда до сусіда, у напрямку, у якому значення хвильової матриці є меншим на одиницю від поточного.

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	4	3	2	3	4	5	6	-1
-1	3	-1	1	-1	5	-1	7	-1
-1	2	1	0	-1	6	7	8	-1
-1	3	-1	-1	-1	-1	-1	9	-1
-1	4	5	6	-1	12	11	10	-1
-1	-1	-1	7	-1	-1	-1	11	-1

-1	10	9	8	9	10	-1	12	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

Завдання для самостійної роботи

Задачі з сайту e-olimp.com:
Алгоритми у лабіринтах: 4001, 1061, 432, 1062

РОЗДІЛ 8. ДИНАМІЧНЕ ПРОГРАМУВАННЯ

§8.1. Поняття про динамічне програмування

СПИСОК ЛІТЕРАТУРИ ТА ВИКОРИСТАНІ ДЖЕРЕЛА

https://www.youtube.com/watch?v=IsaS0NmgXlg
https://www.youtube.com/watch?v=pxR3UoO9c9w

1. [Bruno] Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in Python.

2. Калиткин Н.Н. Численные методы.– Наука, 1978. — 512 с.

Мої програми: https://1drv.ms/f/s!AkQ93-IIgoCHgdhp5ERnssDq46FFCw (Графи – папка Т4)

Лекция 4: Теория графов. https://www.youtube.com/watch?v=npV3mOIZJNc&t=3810s

Лекция 5: Поиск в графах и обход. Алгоритм Дейкстры
https://www.youtube.com/watch?v=IjLHY5U4y2c&t=125s

Алгоритм Дейкстры: https://www.youtube.com/watch?v=tyQSgTytc4s
Алгоритм Дейкстры: https://www.youtube.com/watch?v=UA6aV1XJCGg&t=1019s

[Bruno] Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in Python.

Problem Solving with Algorithms and Data Structures using Python
http://interactivepython.org/runestone/static/pythonds/index.html
http://aliev.me/runestone/index.html (переклад на російську мову попереднього матеріалу)

http://www.mi.ras.ru/~podolskii/files/lecture7.pdf

https://server.179.ru/~yurkov/1011/7b/matprak/11graphs.pdf

Додаток для системи Android, що ілюструє роботу різноманітних алгоритмів.
https://play.google.com/store/apps/details?id=wiki.algorithm.algorithms