Лабораторна робота N2

Івасюк Михайло, Коваль Вікторія

Звіт

Алгоритм Гаффмана

Алгоритм виконаний у парадигмі ООП та у вигляді дерева. Для роботи алгоритму прийшлося написати допоміжний клас “Node”, який має такі атрибути як: символ, частота символа в тексті, лівий син, правий син.

```
class Node:
    def __init__(self, freq, char = None) -> None:
        self.char = char
        self.freq = freq
        self.left_child = None
        self.right_child = None
    def __repr__(self) -> str:
        return f'("{self.char}", {self.freq})'
```

Також необхідно було ще написати функцію яка знаходить частоту появи символів в тексті, вона видає словник де ключ це символ, а значення це частота.

```
def frequency(self, text):
    freq = {}
    for char in text:
        if char not in freq:
            freq[char] = 1
        else:
            freq[char] += 1
    return sorted(freq.items(), key= lambda x: x[1])
```

Кодування тексту: спочатку знаходиться частота появи кожного символа, потім, створюється список *nodes з* обєктами класу *Node*, після того основний цикл *while* з кожною ітерацією видаляє зі списку 2 вершини, *(на першій ітерації по суті всі вершини є листками),* та створює і додає в список *nodes* новий обєкт класу *Node*, де символ - *None*, а частота - це сума

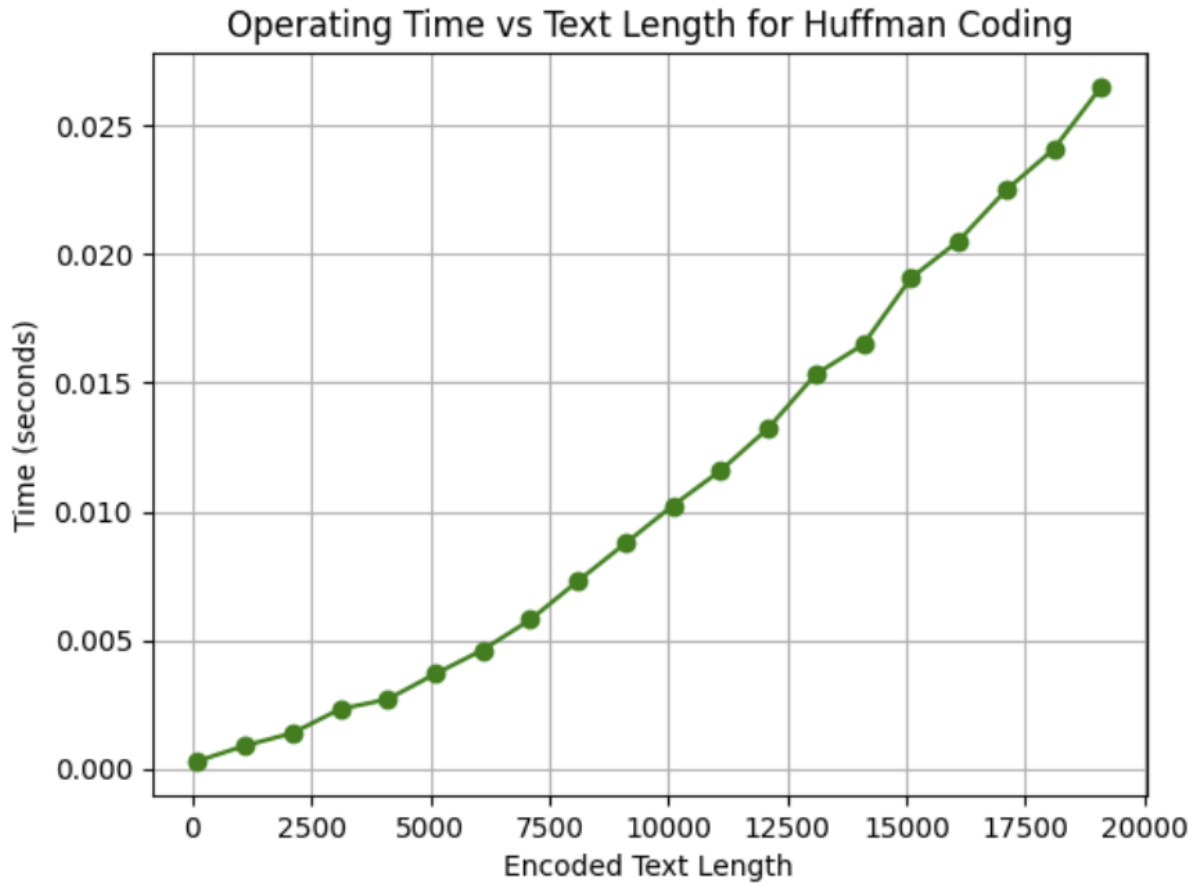
частот двох попередніх вершин, та робить лівого сина першу вершину, а правого сина другу. Після завершення циклу, в нас залишається список з однією вершиною - коренем. Тепер потрібно згенерувати словник. Викликається функція *generate_code*, яка рекурсивно створює словник де ключ це символи а значення - кодування. Після того через цикл створюється закодоване повідомлення. В результаті, повертається закодоване повідомлення та словник.

```
class Huffman:
    main_dict = {}
    def generate_code(self, node: 'Node', coding):
        if node.char is not None:
            char = node.char
            code = coding
            self.main_dict[char] = code
        else:
            self.generate_code(node.left_child, coding + '0')
            self.generate_code(node.right_child, coding + '1')

    def encode(self, text: str) -> tuple[str, dict[str, str]]:
        frequency = self.frequency(text)
        nodes = []
        for (char, freq) in frequency:
            nodes.append(Node(freq, char))
        while len(nodes) > 1:
            first_lowest = nodes[0]
            second_lowest = nodes[1]
            min_freq = nodes.pop(0).freq + nodes.pop(0).freq
            new_node = Node(min_freq)
            new_node.left_child = first_lowest
            new_node.right_child = second_lowest
            nodes.append(new_node)
            nodes = sorted(nodes, key= lambda x: x.freq)
        self.generate_code(nodes[0], '')
        coded_str = ''
        for i in text:
            coded_str += self.main_dict[i]
        return (coded_str, self.main_dict)
```

Декодинг: декодування дуже просте - поки стрічка коду не є пуста, циклом *for* ми заходимо в ключі словника *(попередньо я перевернув словник, тепер ключі це кодування а значення символи)* і якщо стрічка починається з цього кодування, то додаємо значення коду, тобто символ, до стрічки, і видаляєм декодоване значення з стрічки.

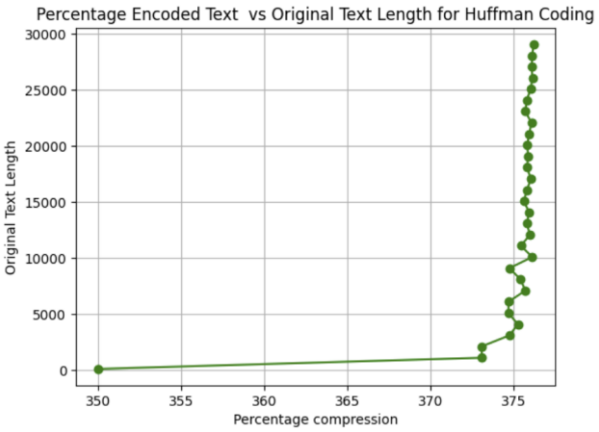
```python
def decode(self, code: str, coding_dict: dict[str, str]):
    decoded_str = ""
    coding_dict = {i : j for j, i in coding_dict.items()}
    while code:
        for cd in coding_dict:
            if code.startswith(cd):
                decoded_str += coding_dict[cd]
                code = code[len(cd):]
    return decoded_str
```

Графік роботи відносно розміру вхідних даних



По графіку можна побачити, що чим більша довжина тексту, понад 10000 символів, то більшим є його середній час роботи. Для довжини стрічки в 20000, середній час виконання складав 0.025 секунд.

Ступінь стиснення відносно розміру вхідних даних (у відсотках), Гаффман



Для цього графіку, я брав відсоток який складає довжина закодованого повідомлення, та віднімав 100%, щоб отримати саме приріст в відсотках, відповідно, отримав такий результат. Загалом, чим більша довжина тексту тим стає меншою різниця приросту відсотків для 2 повідомлень. Відповідно починаючи з повідомлень довжиною 15000, середня довжина закодованого коду збільшується на 375%.

Висновок: алгоритм Гаффмана краще використовувати для стиснення даних у випадках, коли є чіткий дисбаланс у частоті входження символів, тому що чим частіше зустрічається символ, тим коротший його код.

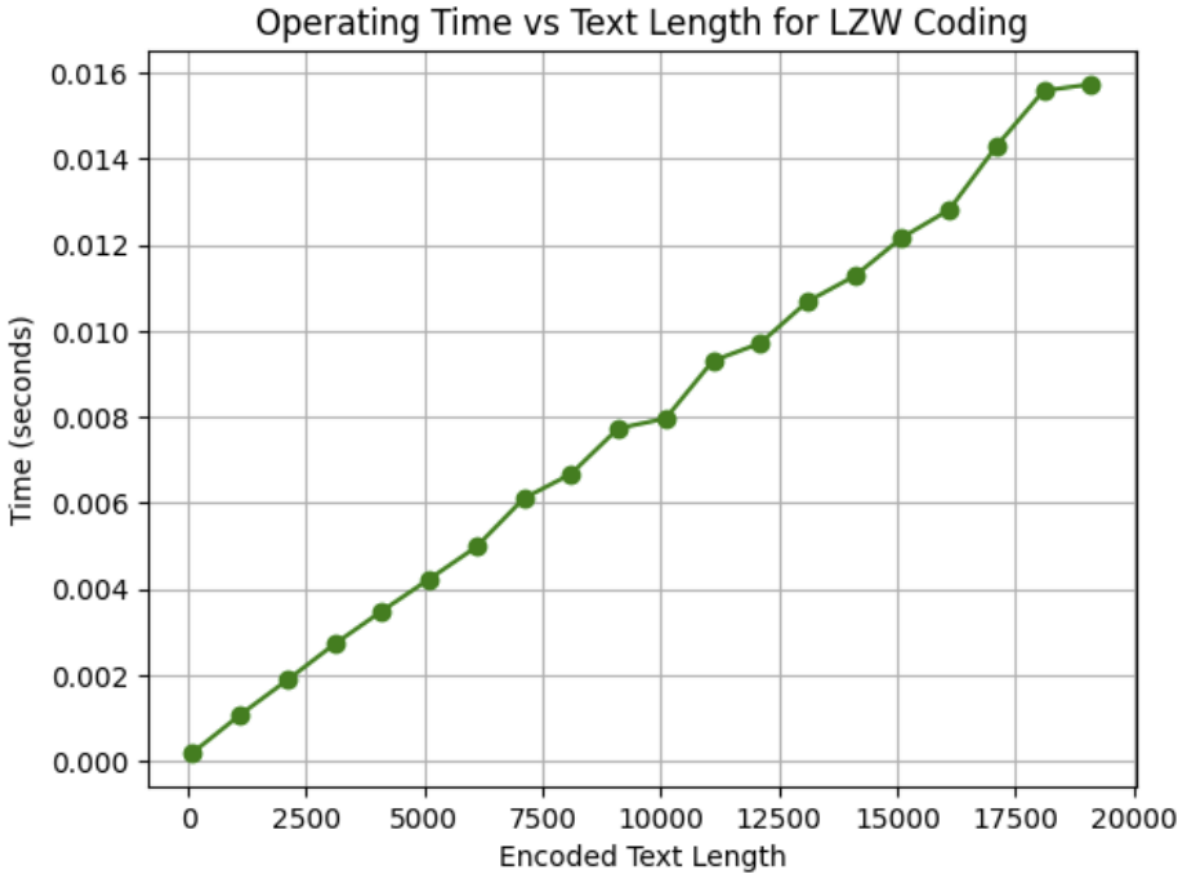
Алгоритм LZW

```python
import copy
class LZW:
    def encode(self, text: str) -> tuple[str, list]:
        encoded_stringa = []
        start_dict = {}
        num = 1
        for char in text:
            if char not in start_dict:
                start_dict[char] = num
                num += 1
        stringa = ''
        dict_use = copy.deepcopy(start_dict)
        for ind, char in enumerate(text):
            stringa += char
            if ind == len(text) - 1:
                encoded_stringa.append((dict_use[stringa]))
                return [encoded_stringa, start_dict]
            new_str = stringa + text[ind + 1]
            if new_str not in dict_use:
                dict_use[new_str] = num
                num += 1
                encoded_stringa.append((dict_use[stringa]))
                stringa = ''
    def decode(self, code: str, coding_dict: list) -> str:
        decoded_text = ''
        last_num = list(coding_dict.values())[-1] + 1
        string = ''
        for indx, num in enumerate(code):
            if num not in list(coding_dict.values()):
                coding_dict[string + string[0]] = num
            decoded_text += str(list(coding_dict.keys())[list(coding_dict.values()).index(num)])
            if len(string) != 0:
                coding_dict[string + ((list(coding_dict.keys())[list(coding_dict.values()).index(num)])[0])] = last_num
                last_num += 1
            string = list(coding_dict.keys())[list(coding_dict.values()).index(num)]
        return decoded_text
```

Кодування: під час кодування, алгоритм проходиться по тексту та додає символи до поточної стрічки, перевіряючи, чи існує вже така послідовність у словнику. Якщо такої послідовності немає, вона додається до словника разом із новим унікальним кодом. Якщо є така послідовність, то цикл далі, продовжується, просто попередньо додавши наступний символ до стрічки.

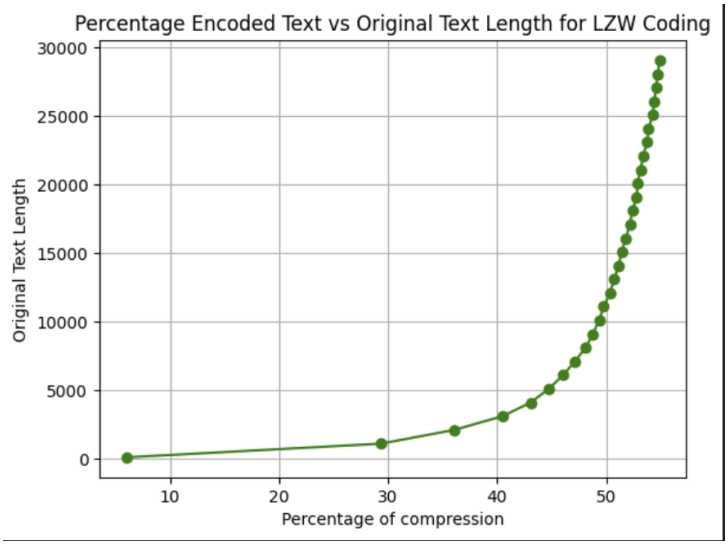
Декодування: На початку програма ініціалізує порожній рядок для зберігання розкодованого тексту *decoded_text*, визначає останній номер у словнику *coding_dict* та порожній рядок для зберігання попереднього рядка *string*. Програма проходиться через кожен символ у стрічці *code*. Для кожного коду вона перевіряє, чи вже існує цей код у словнику. Якщо ні, то програма додає нову пару ключ-значення у словник, де ключ - це попередній рядок, доповнений його першим символом, а значення - код символу. В результаті додає розкодований символ до змінної *decoded_text.*

Графік роботи відносно розміру вхідних даних, LZW



Загалом LZW показав себе набагато краще в швидкості кодування ніж алгоритм Гаффмана. Стрічку довжиною 20000 він кодує за 0.016 секунд, порівнянно з 0.025 секунд за алгоритмом Гаффмана.

Ступінь стиснення відносно розміру вхідних даних (у відсотках), LZW



По графіку можна зробити висновок, що LZW працює краще чим більша довжина стрічки, відповідно для 30000 - алгоритм стиснув текст більше ніж на 50 відсотків. Я вважаю це хорошим результатом та вдалу імплементацію коду.

Висновок: загальною перевагою алгоритму LZW є його ефективність у стисненні текстових даних, особливо які містять повторювані шаблони або фрази.

Алгоритм LZ77

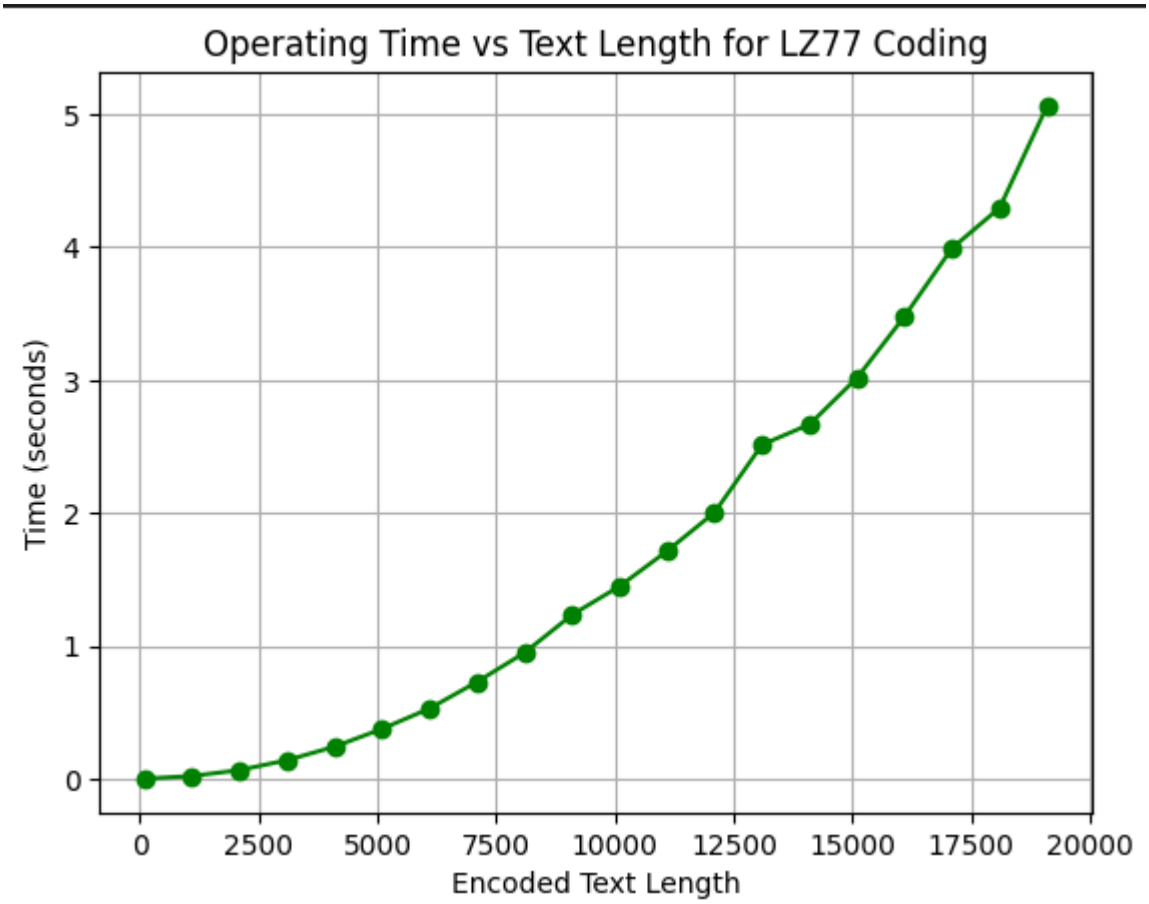
```python
class LZ77:
    def __init__(self, buffer_size: int):
        self.buffer_size=buffer_size
    def _text2list(self,text1):
        return list(text1)
    def _list2text(self,lst):
        return ''.join(lst)
    def encode(self, text: str) -> str:
        lst=self._text2list(text)
        compressed = []
        index = 0
        while index < len(lst):
            best_offset = -1
            best_length = -1
            best_match = ''
            for length in range(1, min(len(lst) - index, self.buffer_size)):
                in_set=lst[index:index + length]
                in_set_str=self._list2text(in_set)
                lst_new_str=self._list2text(lst)
                offset=lst_new_str.rfind(in_set_str,max(0, index - self.buffer_size), index)

                if offset != -1 and length > best_length:
                    best_offset = index - offset
                    best_length = length
                    best_match = in_set
            if best_match:
                compressed.append((best_offset, best_length, lst[index + best_length]))
                index += best_length + 1
            else:
                compressed.append((0, 0, lst[index]))
                index += 1
        return compressed
    def decode(self, code: str) -> str:
        decompressed = []
        for item in code:
            offset=item[0]
            length=item[1]
            next1 = item[2]
            if length == 0:
                decompressed.append(next1)
            else:
                start = len(decompressed) - offset
                in_set= decompressed[start:start + length]
                decompressed.extend(in_set)
                decompressed.append(next1)
        return self._list2text(decompressed)
```

Так як алгоритм LZ77 закодовує текст за принципом <offset,length,next cherecter> зручно використати додаткові функції _text2list і _list2texst де наш list це список кортежів.

Кодування: Спочатку ми перетворюємо наш текст на список з літер, визначаємо пустий список compresed та даємо початкове значення індексу 0. Дальше ми ітеруємось по циклу while поки наш індекс не стане таким як і довжина нашого тексту. Так як наш офсет та довжина це завжди не від'ємне число то зручно спочатку присвоїти їм будь-яке відємне число. Дальше за допомогою циклу for ми шукаємо всі фрагменти тексту які можна стиснути і за допомогою rfind знаходимо останній збіг в рядку з нашим підрядком, який і буде нашим найбільшим зсувом. Якщо довжина наших співпадунь стає більшою за нашу попередню довжину, тоді ми знайшли нову найкращу довжину і оновили наші дані. Дальше іфкою ми перевіряємо чи був знайдений який-небудь збіг фрагментів даних, який можна стиснути. Якщо так то додаємо кількість повторів і довжину збігу, якщо ні то вертаємо кортеж <0,0,новий символ>. По закінченню циклу отримуємо список з кортежами, які містять інформацію про наш закодований текст.

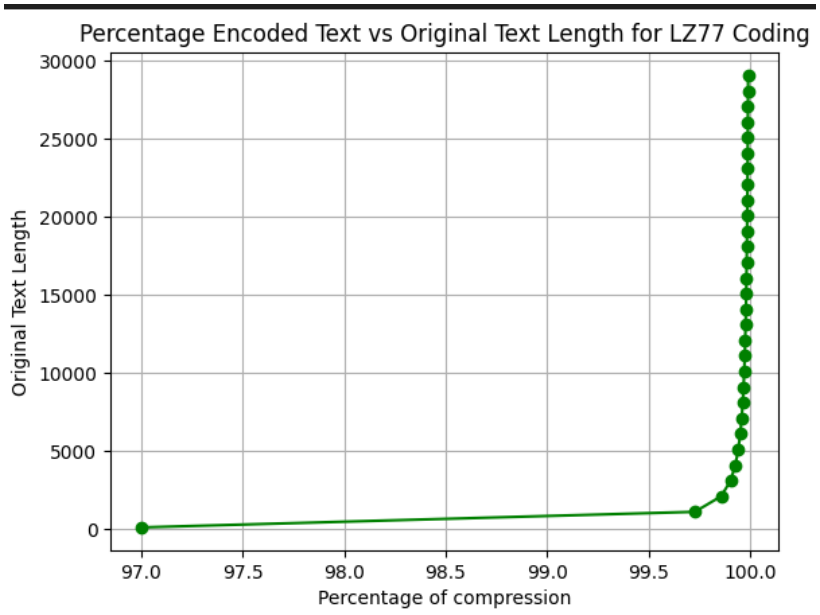
Декодування: Тут ми створюємо список decompresed куди будемо додавати наші розшифровані символи. Циклом for проходимо по кожному кортежу з нашим кодом. Якщо довжина рівна 0 то просто додаємо наш символ в список, якщо не 0 то знаходимо початок фрагмента для розшифрування за допомогою дожини офсету і дістаємо фрагмент вказаної довжини і додаємо символ next1 до розкодованого тексту. Повертаємо строку з наших розкодованих символів.

Графік роботи відносно розміру вхідних даних, LZ77



За графіком ми бачимо, що lz77 не найкращий для кодування так як кодування 20000 символів у нас зайняло цілих 5 секунд, що набагато довше ніж в lzw.

Ступінь стиснення відносно розміру вхідних даних (у відсотках), LZ77



По графіку ми бачимо що lz77 також не ефективну стискає текст так як довжина нашого стиснутого тексту мінімально менша ніж оригінальний текст, тому тут lz77 теж поступається перед LZW.

Висновок: У нашому звіті добре підкреслено, що кожен алгоритм має свої особливості і використовується в залежності від конкретного контексту. Для текстових даних з повторюваними шаблонами найефективнішим виявився LZW, в той час як Гаффман може бути кращим в інших сценаріях з нерівномірним розподілом символів. LZ77, хоча і може здійснювати стиснення, виявився менш ефективним порівняно з іншими алгоритмами.