1. **ИП-3. Колоквиум - 1**

**1. Алгоритмы, виды сложности, виды асимптотических оценок,**

**2. Правила оценки асимптотической сложности**

**3. Рекуррентные уравнения, способы реализации на примере расчета чисел Фибоначчи**

**4. Исследование рекуррентных соотношений на основе дерева рекурсии**

**5. Оценка решения рекуррентного соотношения методом подстановки**

**6. Парадигма «Разделяй и властвуй». Основная теорема о рекуррентных соотношениях.**

**7. Постановка и решение задачи Рюкзака 0-1 методом ДП**

**8. Длинная арифметика. Алгоритм Карацубы.**

**9. Классический алгоритм Хаффмана на примере 4-значных чисел. (нет 4-значных чисел)**

**10. Простейшие структуры данных: очереди, стеки, деки; операции, структуры хранения.**

**11. Интерполяционный поиск в массиве чисел.**

**12. Задача кодирования сообщений, равномерные и неравномерные коды. Сжатие сообщений**

**13. DSA алгоритм, расчет открытого и закрытого ключа**

**14. АТД Бинарная куча. Назначение, операции, применение, алгоритм реализации**

**15. АТД DSU (СНМ), Назначение, операции, применение, алгоритм реализации**

**16. Деревья, термины и определения, основные операции**

**17. N-арные деревья, представление, виды обходов**

**18. Бинарные деревья, виды представлений в памяти, виды обходов, операции добавления и удаления, поиск значения, минимума, максимума, расчет сумм, количества**

**19. Упорядоченное бинарное дерево, поиск значения, минимума, максимума, расчет сумм, количества**

**20. Постановка задачи о кратчайших путях, примеры применения**

**21. Алгоритм Дейкстры на основе сортировки, сложность, ограничения**

**22. Алгоритм Дейкстры на основе приоритетной очереди, сложность, ограничения**

**23. Постановка задачи о минимальном остовном дереве (МОД), примеры применения.**

**24. Алгоритм Прима на основе сортировки, сложность, ограничения**

**25. Алгоритм Прима на основе приоритетной очереди, сложность, ограничения**

**26. Алгоритм Крускала, сложность, ограничения**

**27. Сортировка, назначение, основные виды и их особенности**

**28. Алгоритм Шелла, оценки сложности, устойчивость**

**29. Шейкер-сортировка, оценки сложности, устойчивость**

**30. Сортировка Хоара, оценки сложности, устойчивость**

**31. Пирамидальная сортировка, оценки сложности, устойчивость**

**32. Сортировка подсчетом, оценки сложности, устойчивость**

1. **Волновой алгоритм,пошаговое описание,сложность**

**34. Алгоритм Флойда-Уоршалла, пошаговое описание ,сложность**

1. **Алгоритмы, виды сложности, виды асимптотических оценок,**

**Алгоритм** – точное предписание, которое задает вычислительный процесс, начинающийся с произвольного **исходного данного** и направленный на получение полностью определенного этим исходным данным **результата**.

**Алгоритм** – это четкое описание по выполнению некоторого **процесса обработки данных**, который через разумное конечное число шагов приводит к решению задачи данного типа для любых допустимых вариантов исходных данных.

**Свойства алгоритмов:**

1. **Дискретность (прерывность)** - алгоритм должен представлять процесс решения задачи как последовательное выполнение простых шагов.

2. **Определенность** - каждое правило алгоритма должно быть четким, однозначным и не оставлять места для вариаций.

3. **Результативность (конечность)** - алгоритм должен приводить к решению задачи за конечное число шагов.

4. **Детерминированность**. После каждого шага необходимо указывать, какой шаг выполняется следующим, либо давать команду остановки

5. **Массовость** - алгоритм решения задачи разрабатывается в общем виде и должен быть применим для некоторого класса задач, различающихся только входными данными. При этом входные данные могут выбираться из некоторой области, которая называется **областью применимости алгоритма**.

**Формы представления алгоритмов:**

1. **Словесная** (записи на естественном языке);

2. **Графическая** (изображения из графических символов);

3. **Псевдокоды** (описания алгоритмов на условном алгоритмическом языке, включающие в себя элементы языка программирования, фразы естественного языка, общепринятые математические обозначения и др.);

4. **Программная** (тексты на языках программирования)

**Сложность алгоритмов**

**Анализ трудоёмкости алгоритмов**

Целью анализа трудоёмкости алгоритмов является нахождение оптимального алгоритма для решения данной задачи.



**Теория сложности**, являясь частью **теории вычислений**, изучает ресурсы или стоимость вычислений, необходимые для выполнения поставленной проблемы.

**Вычислительная сложность алгоритма** — это функция, определяющая зависимость объёма работы, выполняемой некоторым алгоритмом, от свойств входных данных. Объём работы обычно измеряется абстрактными понятиями времени и пространства, называемыми вычислительными ресурсами.

**Время** определяется количеством элементарных шагов, необходимых для решения проблемы, тогда как **пространство** определяется объёмом памяти или места на носителе данных.

**Градации сложности**

𝑓 𝑛 = 𝑂(1) – константная;

𝑓 𝑛 = 𝑂(𝑙𝑜𝑔 𝑛) – логарифмическая; (см. пример ^ алгоритм 2)

𝑓 𝑛 = 𝑂 𝑛 – линейная; (см. пример \* алгоритм 2)

𝑓 𝑛 = 𝑂(𝑛 𝑐) - полиномиальная (квадратичная, кубическая …); (см. пример\* алгоритм 1)

𝑓 𝑛 = 𝑂(𝑐 𝑛) – экспоненциальная; (см. пример ~ алгоритм 1)

𝑓 𝑛 = 𝑂(𝑛!) – факториальная

Запись вида f(n) = O(g(n)) означает, что ф-ия f(n) возрастает медленнее чем ф-ия g(n), т.е если при 𝑔 𝑛 > 0 и 𝑛 > 0 существуют положительные с1и 𝑛0, такие что при любом 𝑛 > 𝑛0 выполняется 𝑐1 ∙ 𝑔 𝑛 ≥ 𝑓(𝑛). с1и 𝑛0 могут быть сколь угодно большими числами.

Такая **O оценка** дает нам верхнюю оценку временной трудоемкости алгоритма – его асимптотическую сложность.

1. **Правила оценки асимптотической сложности**

**Оценка сложности**

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

**Подсчет операций. Классы входных данных**

Одним из способов оценки трудоемкости (*Tn*) является подсчет количества выполняемых операций. Рассмотрим в качестве примера алгоритм поиска минимального элемента массива.

**начало; поиск минимального элемента массива array из N элементов**

**min := array[1]**

**для i от 2 до N выполнять:**

**если array[i] < min**

**min := array[i]**

**конец; вернуть min**

При выполнении этого алгоритма будет выполнена:

1. N — 1 операция присваивания счетчику цикла i нового значения;
2. N — 1 операция сравнения счетчика со значением N;
3. N — 1 операция сравнения элемента массива со значением min;
4. от 1 до N операций присваивания значения переменной min.

Точное количество операций будет зависеть от обрабатываемых данных, поэтому имеет смысл говорить о наилучшем, наихудшем и среднем случаях. При этом худшему случаю всегда уделяется особое внимание, в том числе потому, что «плохие» данные могут быть намеренно поданы на вход злоумышленником.

Понятие среднего случая используется для оценки поведения алгоритма с расчетом на то, что наборы данных равновероятны. Однако, такая оценка достаточно сложна:

1. исходные данные разбиваются на группы так, что трудоемкость алгоритма (*ti*) для любого набора данных одной группы одинакова;
2. исходя из доли наборов данных группы в общем числе наборов, рассчитывается вероятность для каждой группы (*pi*);
3. оценка среднего случая вычисляется по формуле:

**O(n) — линейная сложность**

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

**O(log n) — логарифмическая сложность**

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

**O(n2) — квадратичная сложность**

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n \* n, т. е. n2.

1. **Рекуррентные уравнения, способы реализации на примере расчета чисел Фибоначчи**

Рекуррентное уравнение — это уравнение, которое определяет последовательность чисел через предыдущие члены этой последовательности.

Например, рекуррентное уравнение для чисел Фибоначчи выглядит так:

F(n) = F(n-1) + F(n-2), где F(0) = 0 и F(1) = 1.

Это уравнение говорит нам, что каждое число Фибоначчи равно сумме двух предыдущих чисел Фибоначчи.

Существует несколько способов реализации расчета чисел Фибоначчи на основе данного рекуррентного уравнения.

**1. Рекурсивный подход:**

def fibonaccirecursive(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

return fibonaccirecursive(n-1) + fibonaccirecursive(n-2)

Этот подход использует само рекуррентное уравнение для расчета чисел Фибоначчи. Однако, такой подход неэффективен, так как он повторно вычисляет одни и те же значения.

**2. Динамическое программирование:**

def fibonaccidp(n):

fib = 0, 1

for i in range(2, n+1):

fib.append(fibi-1 + fibi-2)

return fibn

Этот подход использует массив для хранения уже вычисленных значений, чтобы избежать повторных вычислений.

**3. Матричный подход:**

import numpy as np

def fibonaccimatrix(n):

A = np.array([[1, 1], [1, 0]])

return np.linalg.matrixpower(A, n)01

Этот подход использует свойства матриц для нахождения значения числа Фибоначчи.

Каждый из этих подходов имеет свои преимущества и недостатки в зависимости от задачи и входных данных.

1. **Исследование рекуррентных соотношений на основе дерева рекурсии**

Основными современными инструментами исследования рекурсивных алгоритмов являются:

1) метод исследования дерева рекурсии.

2) метод анализа рекуррентных соотношений.

Дерево рекурсии — это графическое представление рекурсивного алгоритма, которое позволяет лучше понять его работу и оценить его временную сложность. Исследуя дерево вызова, мы ищем закономерности для вывода рекуррентной формулы и расчет временной сложности.

Для исследования рекуррентных соотношений на основе дерева рекурсии необходимо выполнить следующие шаги:

1. Построить дерево рекурсии для данного рекуррентного соотношения. Каждый узел дерева соответствует вызову рекурсивной функции, а дочерние узлы соответствуют вызовам этой функции изнутри.

2. Оценить количество листьев дерева рекурсии. Листья соответствуют базовым случаям рекурсии, которые не вызывают новых рекурсивных вызовов.

3. Оценить количество узлов на каждом уровне дерева рекурсии. Это позволяет оценить количество рекурсивных вызовов на каждом уровне.

4. Оценить сложность вычисления значения рекуррентного соотношения путем сложения времени выполнения всех узлов дерева рекурсии.

5. Оценить общую временную сложность алгоритма путем умножения сложности одного узла на количество узлов в дереве рекурсии.

6. Проверить полученные результаты, используя математические методы, такие как метод подстановки или метод характеристического уравнения.

Исследование рекуррентных соотношений на основе дерева рекурсии позволяет получить точную оценку временной сложности алгоритма и оптимизировать его, если это возможно.

1. **Оценка решения рекуррентного соотношения методом подстановки**

Метод подстановки — это метод решения рекуррентных уравнений, который заключается в последовательной подстановке выражений для предыдущих членов последовательности в рекуррентное уравнение до тех пор, пока не будет получено выражение для n-го члена последовательности.

Например, для рекуррентного уравнения чисел Фибоначчи F(n) = F(n-1) + F(n-2), можно использовать метод подстановки следующим образом:

F(n) = F(n-1) + F(n-2)

= (F(n-2) + F(n-3)) + (F(n-3) + F(n-4))

= F(n-2) + 2\*F(n-3) + F(n-4)

Затем можно продолжить подстановку для предыдущих членов, пока не будет получено выражение для F(n) только через начальные значения F(0) и F(1).

Оценка решения рекуррентного уравнения методом подстановки может быть сложной и требует определенных математических навыков. Кроме того, этот метод может быть неэффективным для больших значений n, так как требует многократных вычислений предыдущих членов последовательности. Поэтому обычно используются более эффективные алгоритмы, такие как динамическое программирование или матричный подход.

1. **Парадигма «Разделяй и властвуй». Основная теорема о рекуррентных соотношениях.**

Основная теорема о рекуррентных соотношениях — это формула, предназначенная для решения рекуррентных соотношений следующего вида:

T(n) = aT(n/b) + f(n), где

n = объем входных данных

a = количество подзадач в рекурсии

n/b = размер каждой подзадачи. Предполагается, что все подзадачи имеют одинаковый размер.

f(n) = оценка выполненной работы вне рекурсивных вызовов.

Также она включает в себя вычислительную стоимость деления на подзадачи объединения решений этих подзадач.

Здесь a ≥ 1 и b > 1 — константы, а f(n) — асимптотически положительная функция.

Асимптотически положительная функция — функция, где при достаточно больших значениях n f(n)>0.

Основная теорема о рекуррентных соотношениях — простой и быстрый способ вычисления временной сложности рекуррентных соотношений (например, «Разделяй и властвуй»).

1. **Постановка и решение задачи Рюкзака 0-1 методом ДП**

Постановка:

Общий случай задачи:

В общем виде задачу имеет вид: из заданного множества предметов со свойствами «стоимость» и «вес» требуется отобрать подмножество с максимальной полной стоимостью, соблюдая при этом ограничение на суммарный вес.

Это общий случай задачи. Если w – вместимость рюкзака, Wi – вес предмета, а Ci – его ценность, то решение находится в пространстве значений вектора X, таких что:

X\*W <= w

X\*C => max

Для преобразования общей постановки в задачу рюкзаку 0-1 мы добавим условие:

• Xi = 0 – предмет не берем в рюкзак

• Xi = 1 – предмет берем в рюкзак

Если w – вместимость рюкзака, Wi – вес предмета, а Ci – его ценность, то решение находится в пространстве значений вектора X, таких что

X\*W <= w

X\*C => max

Где X – бинарный вектор решения о выборе предметов.

Тогда это задача сводится к поиску битового подмножества X, которое приводит в данному решению.

Решение:

для динамического программирования

Чтобы решить поставленную задачу, требуется решить отдельные части задачи (подзадачи), после чего объединить решения подзадач в одно общее решение.

• Решить каждую подзадачу только один раз, сократив тем самым количество вычислений.

Пусть w – вместимость рюкзака.

Введем понятие «Потенциального рюкзака s».

Это рюкзак, вместимостью s.

Нам он необходим для сведения задачи к подзадачам.

Пусть A(k,s) есть максимальная стоимость предметов, которые можно уложить в рюкзак вместимости s, если можно использовать только первые k предметов, то есть {n1,n2,…,nk}, назовем этот набор допустимых предметов для рюкзака вместимостью s.

Введем для граничные условия:

- в рюкзак вместимостью 0 ничего положить нельзя

- если не выбрали предмет, то во всех рюкзаках его нет

Далее важный переход.

Мы последовательно (от 1..n) просматриваем предметы, пытаясь их разместить в рюкзаке.

• Пусть на (к-1)-м шаге мы разместили предметы (от 1..(k-1)) в рюкзаках 0,..,s,..,w.

• Пусть мы также уже на к-м шаге (выбора предметов) выполнили размещение для рюкзаков 1..(s-1).

• Для s-го рюкзака возможны два случая:

1) Предмет k не попадает в рюкзак.

Тогда A(k, s) равно максимальной стоимости рюкзака с такой же вместимостью и набором допустимых предметов {n1,n2,…,nk−1} для этого рюкзака. Это решение было найдено на предыдущем шаге, то есть A(k,s)=A(k−1,s).

2) предмет попадает в рюкзак. В этом случае выбирается лучшее решение из двух значений:

- A(k−1,s) - решение для рюкзака s, найденное на предыдущем (к-1) шаге выбора предмета

- A(k−1, s-wk)+sk, либо нового значения, полученного на основе замены на этот предмет. Мы также изменяем значение стоимости

Лучшее – это лучшее по максимуму. Поэтому берем так:

A(k,s) = max( A(k−1,s), A(k−1,s−wk)+sk ), где стоимость в рюкзаке s на шаге равна максимуму из двух значений

По завершению всех шагов стоимость искомого набора равна A(n,W), так как нужно найти максимальную стоимость рюкзака, где все предметы проверены на допустимость и вместимость рюкзака W.

1. **Длинная арифметика. Алгоритм Карацубы.**

Алгоритм Карацубы — это алгоритм быстрого умножения, разработанный М.Карацубой в 1960-х годах. Он позволяет умножать два n-значных числа за O(n^log2(3)) времени, что быстрее, чем классический алгоритм умножения, который имеет сложность O(n^2).

Алгоритм Карацубы основан на разложении умножаемых чисел на две половины и рекурсивном вычислении их произведения. Этот алгоритм использует идею того, что умножение двух n-значных чисел можно выразить в виде:

A \* B = (a \* 10^(n/2) + b) \* (c \* 10^(n/2) + d)

где a и c - верхние половины чисел A и B, b и d - нижние половины чисел A и B.

Раскрыв эту формулу, мы можем записать умножение A и B как:

A \* B = ac \* 10^n + ((a + b) \* (c + d) - ac - bd) \* 10^(n/2) + bd

Это уравнение показывает, что умножение A и B может быть разбито на три произведения: ac, bd и (a + b) \* (c + d). Последнее произведение может быть вычислено рекурсивно с использованием этого же алгоритма. Таким образом, мы можем получить произведение A и B, используя только три произведения меньшего размера, что позволяет уменьшить количество операций умножения.

Пример:

Для удобства будем использовать десятичную систему(x^10kx=10^k вместо x=2^k

1213 \* 2311:

12+13 =25, 23+11 =34; 25\*34

2+5=7, 3+4 =7; 7\*7 = 39

2\*3 =6; 5\*4 =20

25\*34= 6\*10^2+(49-6-20)\*10+20=850

12\*23

1+2=3,2+3=5;3\*5=15

1\*2=2;2\*3=6;

12\*23 = 2\*10^2+(15-2-6)\*10+6 =276

13\*11

1+3 =4, 1+1=2; 4\*2=8

1\*1=1;3\*1 =3

Результат:

1213\*2311=276\*10^4+(850-276-143)\*10^2+143 =2803243

1. **Классический алгоритм Хаффмана на примере 4-значных чисел.**

Алгоритм Хаффмана — это алгоритм сжатия данных, который используется для уменьшения объема передаваемой информации. Он основан на построении бинарного дерева, в котором каждый лист соответствует символу, а путь от корня до листа представляет код символа. Часто встречающиеся символы кодируются более короткими кодами, что позволяет сократить объем передаваемых данных.

Реализация на Python:

Для начала создадим класс Node, который будет представлять узел дерева:

class Node:

def \_\_init\_\_(self, freq, symbol=None):

self.freq = freq

self.symbol = symbol

self.left = None

self.right = None

def is\_leaf(self):

return self.left is None and self.right is None

Класс Node содержит поля freq (частота символа) и symbol (сам символ), а также указатели на левого и правого потомков. Метод isleaf() проверяет, является ли узел листом.

Теперь напишем функцию buildhuffmantree(), которая будет строить дерево Хаффмана:

```python

def buildhuffmantree(freqdict):

nodes = Node(freq=freq, symbol=symbol) for symbol, freq in freq\_dict.items()

while len(nodes) > 1:

nodes.sort(key=lambda x: x.freq)

left = nodes.pop(0)

right = nodes.pop(0)

parent = Node(freq=left.freq + right.freq)

parent.left = left

parent.right = right

nodes.append(parent)

return nodes0

Функция принимает словарь freq\_dict, содержащий символы и их частоты. Сначала создаются узлы дерева для каждого символа, затем они объединяются в одно дерево путем последовательного объединения двух узлов с наименьшей частотой. Результатом функции является корень дерева.

Теперь напишем функцию encode\_huffman\_tree(), которая будет кодировать символы в дереве:

python

def encodehuffmantree(root):

def encodehelper(node, code):

if node is None:

return {}

if node.isleaf():

return {node.symbol: code}

codes = {}

codes.update(encodehelper(node.left, code + '0'))

codes.update(encodehelper(node.right, code + '1'))

return codes

return encodehelper(root, '')

Функция принимает корень дерева и возвращает словарь, содержащий коды символов. Функция encodehelper() рекурсивно проходит по всем узлам дерева, начиная с корня. Если узел является листом, то он кодируется символом и его кодом. Если узел не является листом, то функция вызывается рекурсивно для его левого и правого потомков, добавляя к коду 0 для левого потомка и 1 для правого потомка.

Наконец, напишем функции encodehuffman() и decodehuffman(), которые будут кодировать и декодировать сообщение соответственно:

def encode\_huffman(message):

freq\_dict = {symbol: message.count(symbol) for symbol in set(message)}

root = build\_huffman\_tree(freq\_dict)

codes = encode\_huffman\_tree(root)

encoded\_message = ''.join([codes[symbol] for symbol in message])

return encoded\_message, root

def decode\_huffman(encoded\_message, root):

decoded\_message = ''

node = root

for bit in encoded\_message:

if bit == '0':

node = node.left

else:

node = node.right

if node.is\_leaf():

decoded\_message += node.symbol

node = root

return decoded\_message

Функция encodehuffman() принимает сообщение и строит дерево Хаффмана, кодирует сообщение с помощью функции encodehuffmantree() и возвращает закодированное сообщение и корень дерева.

Функция decodehuffman() принимает закодированное сообщение и корень дерева, декодирует сообщение, начиная с корня дерева и переходя к левому потомку при встрече 0 и к правому потомку при встрече 1, пока не будет достигнут символ.

1. **Простейшие структуры данных: очереди, стеки, деки; операции, структуры хранения.**

**Простейшие структуры данных** — это основные типы данных, которые используются в программировании для хранения и организации данных. Такие структуры данных, как очереди, стеки и деки, являются важными инструментами в различных областях программирования.

**Стек** — это структура данных, в которой элементы добавляются и удаляются только с одного конца, называемого вершиной стека. Операции добавления элемента в стек и удаления элемента из стека называются соответственно push и pop. Стек работает по принципу "последним пришел - первым ушел" (LIFO - Last In First Out). Для хранения стека обычно используется массив или связный список.

**Очередь** — это структура данных, в которой элементы добавляются в одном конце, называемом "хвост", а удаляются из другого конца, называемого "головой". Операции добавления элемента в очередь и удаления элемента из очереди называются соответственно enqueue и dequeue. Очередь работает по принципу "первым пришел - первым ушел" (FIFO - First In First Out). Для хранения очереди также может использоваться массив или связный список.

**Дек (двусторонняя очередь)** — это структура данных, которая позволяет добавлять и удалять элементы как с головы, так и с хвоста. Для этого у дека есть два указателя: на голову и на хвост. Операции добавления и удаления элемента могут быть выполнены как с головы, так и с хвоста. Дек может использоваться как очередь или как стек.

**Операции, выполняемые над стеком**, очередью или деком, могут варьироваться в зависимости от конкретной задачи. В общем случае, кроме основных операций добавления и удаления элемента, могут также поддерживаться операции проверки наличия элемента, получения значения элемента без удаления и т.д.

**Для хранения стека**, очереди или дека могут использоваться различные структуры данных, например, массив, связный список, кольцевой буфер и т.д. Каждая из этих структур имеет свои преимущества и недостатки и может быть выбрана в зависимости от конкретной задачи и требований к производительности.

1. **Интерполяционный поиск в массиве чисел.**

Интерполяционный поиск — это метод поиска элемента в упорядоченном массиве, который использует интерполяцию для прогнозирования приблизительного местоположения искомого элемента.

Основная идея заключается в том, чтобы определить приблизительную позицию искомого элемента в массиве, исходя из значения элементов массива и интерполированной формулы, и затем сузить диапазон поиска до более узкой области.

Алгоритм интерполяционного поиска может быть описан следующим образом:

1. Извлечь начальный и конечный индексы диапазона поиска (обычно 0 и n-1, где n - длина массива).

2. Вычислить позицию mid, используя формулу интерполяции: mid = low + ((high - low) / (arr[high] - arr[low])) \* (x - arr[low]), где x - значение искомого элемента, arr - упорядоченный массив, low и high - начальный и конечный индексы диапазона поиска.

3. Если arr[mid] равен искомому элементу x, то элемент найден.

4. Если arr[mid] меньше x, то искомый элемент может находиться только в правой части массива, поэтому сузить диапазон поиска до mid+1 и high.

5. Если arr[mid] больше x, то искомый элемент может находиться только в левой части массива, поэтому сузить диапазон поиска до low и mid-1.

6. Повторять шаги 2-5 до тех пор, пока элемент не будет найден или диапазон поиска не будет уменьшен до пустого.

Интерполяционный поиск может быть эффективным, если элементы массива распределены равномерно и имеют постоянный шаг между значениями. В противном случае, если распределение элементов неравномерное, интерполяционный поиск может работать хуже, чем более простые методы, такие как двоичный поиск.

1. **Задача кодирования сообщений, равномерные и неравномерные коды. Сжатие сообщений**

Закодировать текст – значит сопоставить ему другой текст. Кодирование применяется при передаче данных – для того, чтобы сохранить конфиденциальность, чтобы сделать передачу данных более надежной, при том, что канал передачи данных может передавать только ограниченный набор символов (например, — только два символа, 0 и 1) и по другим причинам.

При кодировании заранее определяют алфавит, в котором записаны исходные тексты (исходный алфавит) и алфавит, в котором записаны закодированные тексты (коды), этот алфавит называется кодовым алфавитом. В качестве кодового алфавита часто используют двоичный алфавит, состоящий из двух символов (битов) 0 и 1. Слова в двоичном алфавите иногда называют битовыми последовательностями.

**Равномерное кодирование**

Наиболее простой способ кодирования – побуквенный. При побуквенном кодировании каждому символу из исходного алфавита сопоставляется кодовое слово – слово в кодовом алфавите. Иногда вместо «кодовое слово буквы» говорят просто «код буквы». При побуквенном кодировании текста коды всех символов записываются подряд, без разделителей.

**Неравномерное кодирование**

Равномерное кодирование удобно для декодирования. Однако часто применяют и неравномерные коды, т.е. коды с различной длиной кодовых слов. Это полезно, когда в исходном тексте разные буквы встречаются с разной частотой. Тогда часто встречающиеся символы стоит кодировать более короткими словами, а редкие – более длинными. В отличие от равномерных кодов не все неравномерные коды допускают однозначное декодирование.

Есть простое условие, при выполнении которого неравномерный код допускает однозначное декодирование.

**Код называется префиксным**, если в нем нет ни одного кодового слова, которое было бы началом (по-научному, — префиксом) другого кодового слова.

**Сжатие сообщений** представляет собой процесс преобразования исходного сообщения из одной кодовой системы в другую, в результате которого уменьшается *размер сообщения*. Алгоритмы, предназначенные для сжатия информации, можно разделить на две большие группы: реализующие сжатие без потерь (обратимое сжатие) и реализующие сжатие с потерями (необратимое сжатие).

***Обратимое сжатие*** подразумевает абсолютно точное восстановление данных после декодирования и может применяться для сжатия любой информации. Оно всегда приводит к снижению объема выходного потока информации без изменения его информативности, то есть без потери информационной структуры. Более того, из выходного потока, при помощи восстанавливающего или декомпрессирующего алгоритма, можно получить *входной*, а процесс восстановления называется **декомпрессией или распаковкой** и только после процесса распаковки данные пригодны для обработки в соответствии с их внутренним форматом. Сжатие без потерь применяется для текстов, исполняемых файлов, высококачественного звука и графики.

***Необратимое сжатие*** имеет обычно гораздо более высокую степень сжатия, чем *кодирование* без потерь, но допускает некоторые отклонения декодированных данных от исходных. На практике существует широкий круг практических задач, в которых соблюдение требования точного восстановления исходной информации после декомпрессии не является обязательным. Это, в частности, относится к сжатию звука, фото- или видеоизображений. Необратимое сжатие обычно не используется совместно с криптографическим шифрованием, так как основным требованием к криптосистеме является идентичность расшифрованных данных исходным.

Наиболее известный простой подход и *алгоритм* сжатия информации обратимым путем – это *кодирование* серий последовательностей (Run *Length Encoding* – *RLE*). Суть методов данного подхода состоит в замене цепочек или серий повторяющихся байтов на один кодирующий *байт*-заполнитель и *счетчик* числа их повторений. Например, заменять 111111 на одну букву А.

Одними из широко используемых на практике являются словарные методы, к основным представителям которых относятся алгоритмы семейства Зива и Лемпела. Их основная идея заключается в том, что фрагменты входного потока ("фразы") заменяются указателем на то *место*, где они в тексте уже ранее появлялись. В литературе подобные алгоритмы обозначаются как алгоритмы *LZ сжатия*.

Другим подходом к сжатию информации является *код Хаффмана*. Идея алгоритма состоит в следующем: зная вероятности вхождения символов в сообщение, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью присваиваются более короткие коды, тогда как реже встречающимся символам – более длинные. За счет этого достигается сокращение средней длины кодового слова и большая эффективность сжатия. Коды Хаффмана имеют уникальный *префикс* (начало кодового слова), что и позволяет однозначно их декодировать, несмотря на их переменную длину.

1. **DSA алгоритм, расчет открытого и закрытого ключа**

DSA (Digital Signature Algorithm) — это криптографический алгоритм, используемый для создания и проверки цифровой подписи. Он был разработан в 1991 году и является одним из стандартов цифровой подписи в США.

Генерация ключей DSA происходит в несколько этапов:

1)Генерация простого числа p заданной длины в битах. Длина p должна быть кратна 64 битам и находиться в диапазоне от 512 до 1024 бит.

2)Выбор числа q, которое является делителем числа p - 1 и имеет длину в 160 бит.

3)Выбор числа g, которое является первообразным корнем по модулю p и имеет порядок q.

4)Генерация случайного числа x в диапазоне от 1 до q.

5)Вычисление числа y = g^x mod p.

Теперь у нас есть закрытый ключ x и открытый ключ (p, q, g, y), который можно использовать для создания и проверки цифровой подписи.

Создание цифровой подписи DSA происходит в следующих этапах:

1)Выбор случайного числа k в диапазоне от 1 до q.

2)Вычисление числа r = (g^k mod p) mod q.

3)Вычисление числа s = (k^-1 (H(m) + xr)) mod q, где H(m) - хэш сообщения m.

4)Цифровая подпись сообщения m равна паре (r, s).

Проверка цифровой подписи DSA происходит в следующих этапах:

1)Проверка, что r и s находятся в диапазоне от 1 до q.

2)Вычисление хэша сообщения H(m).

3)Вычисление числа w = s^-1 mod q.

4)Вычисление числа u1 = (H(m)w) mod q и u2 = (rw) mod q.

5)Вычисление числа v = ((g^u1 \* y^u2) mod p) mod q.

6)Если v = r, то цифровая подпись является действительной.

1. **АТД Бинарная куча. Назначение, операции, применение, алгоритм реализации**

Алгоритм реализации абстрактного типа данных бинарная куча:

1. Создать массив для хранения элементов кучи.

2. Реализовать операцию вставки элемента в кучу, добавляя элемент в конец массива и перестраивая кучу сверху вниз до тех пор, пока не будет удовлетворяться условие кучи.

3. Реализовать операцию удаления максимального элемента из кучи, заменив его на последний элемент в массиве и перестроив кучу снизу вверх до тех пор, пока не будет удовлетворяться условие кучи.

4. Реализовать операцию поиска максимального элемента в куче без его удаления.

Назначение бинарной кучи заключается в поддержании структуры данных, где каждый узел имеет значение не меньше (для максимальной кучи) или не больше (для минимальной кучи) значений своих потомков. Они широко используются в алгоритмах сортировки, поиска и оптимизации.

Основные операции, которые можно выполнять с бинарной кучей, это вставка элемента в кучу, удаление максимального элемента из кучи и поиск максимального элемента в куче. Операция вставки позволяет добавлять новый элемент в кучу, операция удаления максимального элемента позволяет извлекать наибольший элемент из кучи, а операция поиска максимального элемента позволяет получить значение наибольшего элемента в куче без его удаления.

Применение бинарной кучи может быть различным. Например, они могут использоваться для реализации алгоритма сортировки кучей, где элементы массива добавляются в кучу и извлекаются в порядке возрастания (для минимальной кучи) или убывания (для максимальной кучи). Также они могут использоваться для поиска k-го наибольшего (наименьшего) элемента в массиве, где элементы добавляются в кучу и извлекаются до тех пор, пока не будет найден k-й наибольший (наименьший) элемент.

1. **АТД DSU (СНМ), Назначение, операции, применение, алгоритм реализации**

Алгоритм реализации систем непересекающихся множеств:

1. Создать массив родителей для каждого элемента множества.

2. Инициализировать каждый элемент как отдельное множество, установив его родителя на самого себя.

3. Реализовать операцию объединения двух множеств, найдя их корни и присоединив одно к другому.

4. Реализовать операцию поиска корня множества для проверки принадлежности элемента к множеству.

Назначение систем непересекающихся множеств заключается в группировании элементов в несколько множеств, где каждый элемент может принадлежать только одному множеству. Они широко используются в различных областях, таких как компьютерная графика, анализ данных, алгоритмы оптимизации и т.д.

Основные операции, которые можно выполнять с системами непересекающихся множеств, это объединение двух множеств и поиск корня множества. Операция объединения позволяет объединять два множества в одно, а операция поиска корня позволяет проверять принадлежность элемента к конкретному множеству.

Применение систем непересекающихся множеств может быть различным. Например, они могут использоваться для сжатия данных, где элементы с одинаковым значением могут быть сгруппированы в одно множество. Также они могут использоваться для оптимизации алгоритмов поиска и сортировки данных, где элементы могут быть разбиты на несколько множеств для более эффективной обработки.

void make\_set (int v) {

parent[v] = v;

}

int find\_set (int v) {

if (v == parent[v])

return v;

return find\_set (parent[v]);

}

void union\_sets (int a, int b) {

a = find\_set (a);

b = find\_set (b);

if (a != b)

parent[b] = a;

}

1. **Деревья, термины и определения, основные операции**

Дерево в информатике — это структура данных, которая состоит из узлов, связанных между собой ребрами. Узел представляет собой объект, который может содержать какую-то информацию, а ребро представляет отношение между двумя узлами.

Основные термины и определения:

1) Корень (root) — это узел дерева, не имеющий родителя. Он является началом всего дерева.

2) Ребро (edge) — это связь между двумя узлами дерева.

3) Узел (node) — это элемент дерева, который может содержать некоторую информацию, например, ключ, значение или ссылку на другой узел.

4) Лист (leaf) — это узел, не имеющий дочерних элементов.

5) Родитель (parent) — это узел, который имеет ссылку на другой узел.

6)Дочерний элемент (child) — это узел, на который имеет ссылку родительский узел.

7) Уровень (level) — это количество ребер на пути от корня до данного узла.

8)Высота (height) — это максимальное количество ребер на пути от корня до листа.

Основные операции с деревьями:

• Вставка (insert) - добавление нового узла в дерево.

• Удаление (delete) - удаление узла из дерева.

• Поиск (search) - поиск узла в дереве по ключу.

• Обход (traversal) - обход всех узлов дерева с определенным порядком.

Прямой обход (preorder traversal) - обход узлов сначала корня, затем левого поддерева, затем правого поддерева.

Симметричный обход (inorder traversal) - обход узлов сначала левого поддерева, затем корня, затем правого поддерева.

Обратный обход (postorder traversal) - обход узлов сначала левого поддерева, затем правого поддерева, затем корня.

Деревья используются для решения многих задач, включая поиск, сортировку, хранение и структурирование данных. Каждый вид дерева имеет свои особенности и может использоваться для определенных задач.

Поддерево

• Часть дерева, которая может быть представлена в виде отдельного дерева.

• Степенью вершины в дереве называется количество дуг, которое из нее выходит.

• Степень дерева равна максимальной степени вершин, входящих в дерево. (При этом листьями в дереве являются вершины, имеющие нулевую степень)

1. **N-арные деревья, представление, виды обходов**

В теории графов, дерево – это связный (ориентированный или неориентированный) граф, не содержащий циклов. Т.е. для любой вершины есть один и только один способ добраться до любой другой вершины. В программировании наиболее часто используются бинарные деревья, в которых число исходящих рёбер не превосходит 2, и N-арные деревья с произвольным количеством исходящих ребер.

Бинарное дерево — это конечное множество элементов, которое либо пусто, либо содержит элемент (корень), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями. Каждый элемент бинарного дерева называется узлом. Связи между узлами дерева называются его ветвями. N-арное дерево – это дерево, в котором степени вершин не превосходят N + 1.

В памяти компьютера деревья обычно представляют в виде связной структуры, где каждый узел помимо ключа (key) хранит указатели на дочерние узлы и иногда на родительский. Для хранения N-арных деревьев используют структуру с левым дочерним и правыми сестринским узлами (left-child, right-sibling representation). В этом случае вместо указателя на дочерние узлы каждый узел x хранит два указателя:

•в lef t\_child[x]– указатель на крайний левый дочерний узел узла x;

•в right\_sibling[x]– указатель на узел, расположенный на одном уровне с x справа от него.

Деревья можно обходить «в глубину» или «в ширину». Существует три основных способа обхода «в глубину»:

• прямой (pre-order)

• центрированный (in-order)

inorder(node)

if (node = null)

return

inorder(node.left)

visit(node)

inorder(node.right) iterativeInorder(node)

s ← empty stack

while (not s.isEmpty() or node ≠ null)

if (node ≠ null)

s.push(node)

node ← node.left

else

node ← s.pop()

visit(node)

node ← node.right

• обратный (post-order)

Деревья можно обходить также в порядке уровней, где мы посещаем каждый узел на уровне, прежде чем перейти на следующий уровень. Такой поиск называется поиском в ширину (breadth-first search, BFS).

Обход n-арного дерева. Алгоритмы обхода n-арного дерева.

1)Прямой обход - Посещается корень дерева, затем все узлы левого поддерева, затем узлы след поддеревьев до правого (корень-лево-право).

2)Обратный обход - Сначала все узлы левого поддерева, затем последовательно узлы остальных поддеревьев в обратном порядке, последним посещается корень (лево-прово-корень).

3)Симметричный обход – в симметричном порядке посещаются все узлы левого поддерева, потом корень, затем последовательно в симметричном порядке все узлы остальных поддеревьев (лево-корень-право).

4)Рекурсивный обход дерева(1)

**Прямой**:Preoder(n: узел);

1.Список обхода <- n(Обращение к узлу);

2.For каждого сына m-узла n в порядке слева направо doPreoder(m); (Рекурсивный обход левого поддерева. рекурсивный обход правого поддерева.)

**Обратный**: меняем местами пункты 1 и 2:

1. Рекурсивный обратный обход левого поддерева.

2.Рекурсивный обратный обход правого поддерева.

3. Обращение к узлу.

1. **Бинарные деревья, виды представлений в памяти, виды обходов, операции добавления и удаления, поиск значения, минимума, максимума, расчет сумм, количества**

Бинарное дерево - это структура данных, которая состоит из узлов, каждый из которых имеет не более двух потомков. Каждый узел содержит значение и ссылки на левого и правого потомка.

Представление в виде двух массивов — Left и Right:

o если вершина j является левым (правым) сыном вершины i, то Left[i] = j (Right[i] = j);

o если у вершины i нет левого (правого) сына, то Left[i] = 0 (Right[i] = 0).

А также бинарное дерево можно представить в памяти с помощью класса Node, который будет содержать ссылки на левый и правый узлы, а также значение:

class Node:

def \_\_init\_\_(self, value):

self.left = None

self.right = None

self.value = value

Виды обходов бинарного дерева:

1. Прямой обход (pre-order traversal): сначала посещается корень, затем левое поддерево и правое поддерево.

2. Симметричный обход (in-order traversal): сначала посещается левое поддерево, затем корень и правое поддерево.

3. Обратный обход (post-order traversal): сначала посещается левое поддерево, затем правое поддерево и корень.

Операция добавления элемента в бинарное дерево может быть реализована следующим образом:

def insert(root, value):

if root is None:

return Node(value)

if value < root.value:

root.left = insert(root.left, value)

else:

root.right = insert(root.right, value)

return root

Операция удаления элемента может быть сложной и зависит от конкретной реализации.

Операция поиска значения в бинарном дереве может быть реализована следующим образом:

def search(root, value):

if root is None or root.value == value:

return root

if value < root.value:

return search(root.left, value)

else:

return search(root.right, value)

Операции нахождения минимума и максимума могут быть реализованы следующим образом:

def find\_min(root):

if root is None:

return None

while root.left is not None:

root = root.left

return root.value

def find\_max(root):

if root is None:

return None

while root.right is not None:

root = root.right

return root.value

Операция расчета суммы всех значений в бинарном дереве может быть реализована следующим образом:

def sum\_values(root):

if root is None:

return 0

return root.value + sum\_values(root.left) + sum\_values(root.right)

Количество всех вершин

def count\_nodes(root):

if root is None:

return 0

return 1 + count\_nodes(root.left) + count\_nodes(root.right)

1. **Упорядоченное бинарное дерево, поиск значения, минимума, максимума, расчет сумм, количества**

Бинарное дерево (англ. binary tree) — это упорядоченное корневое дерево, у каждой вершины которого имеется не более двух сыновей. В бинарном дереве каждый сын произвольной вершины определяется как левый или правый.

**Поиск элемента**

Node search(x : Node, k : T):

if x == null or k == x.key

return x

if k < x.key

return search(x.left, k)

else

return search(x.right, k)

**Поиск минимума и максимума**

Node minimum(x : Node):

if x.left == null

return x

return minimum(x.left)

Node maximum(x : Node):

if x.right == null

return x

return maximum(x.right)

**Рассчет сумм**

def sum\_tree(root):

if root is None:

return 0

return root.val + sum\_tree(root.left) + sum\_tree(root.right)

**Рассчет количества элементов**

def count\_nodes(root):

if root is None:

return 0

return 1 + count\_nodes(root.left) + count\_nodes(root.right)

1. **Постановка задачи о кратчайших путях, примеры применения**

Задача о кратчайших путях в графе заключается в нахождении пути минимальной длины между двумя вершинами графа или нахождении кратчайших путей от одной вершины до всех остальных вершин графа.

Примеры применения задачи о кратчайших путях в графе включают:

1. Маршрутизация в компьютерных сетях: задача о кратчайших путях может быть использована для определения оптимального маршрута передачи данных между компьютерами в сети.

2. Поиск кратчайшего пути в навигационных системах: задача о кратчайших путях может быть использована для определения кратчайшего пути между двумя точками на карте.

3. Расписание авиарейсов: задача о кратчайших путях может быть использована для определения оптимального расписания авиарейсов между городами.

4. Проектирование транспортной инфраструктуры: задача о кратчайших путях может быть использована для определения оптимального маршрута для дорожного или железнодорожного транспорта.

5. Анализ социальных сетей: задача о кратчайших путях может быть использована для определения кратчайшего пути между людьми в социальной сети или для определения наиболее влиятельных узлов в сети.

1. **Алгоритм Дейкстры на основе сортировки, сложность, ограничения**

Алгоритм Дейкстры на основе сортировки (Dijkstra's algorithm with heap-based priority queue) является одним из самых эффективных алгоритмов нахождения кратчайшего пути в графе. Он основан на использовании приоритетной очереди (heap-based priority queue), которая позволяет хранить вершины графа в отсортированном порядке по их расстоянию от начальной вершины.

Алгоритм Дейкстры работает следующим образом:

1. Инициализируем расстояния до всех вершин графа как бесконечность, за исключением начальной вершины, расстояние до которой равно 0.

2. Создаем пустую приоритетную очередь и добавляем в нее начальную вершину.

3. Пока приоритетная очередь не пуста:

a. Извлекаем из очереди вершину с наименьшим расстоянием.

b. Для каждого соседа этой вершины:

i. Вычисляем новое расстояние от начальной вершины до этого соседа через текущую вершину.

ii. Если новое расстояние меньше текущего расстояния до соседа, то обновляем расстояние и добавляем соседа в приоритетную очередь.

4. Когда приоритетная очередь пуста, расстояния до всех вершин графа будут найдены.

Сложность алгоритма Дейкстры на основе сортировки зависит от реализации приоритетной очереди. Если использовать бинарную кучу (binary heap) в качестве приоритетной очереди, то сложность алгоритма будет O(E log V), где E - количество ребер в графе, V - количество вершин. Если использовать Фибоначчиеву кучу (Fibonacci heap), то сложность будет O(E + V log V), что является более эффективным.

Ограничения алгоритма Дейкстры на основе сортировки заключаются в том, что он работает только с неотрицательными весами ребер и не может обрабатывать графы с отрицательными циклами. Если в графе есть отрицательный цикл, то алгоритм может зациклиться и не найти кратчайший путь. Для работы с графами с отрицательными весами ребер можно использовать алгоритм Беллмана-Форда (Bellman-Ford algorithm).

1. **Алгоритм Дейкстры на основе приоритетной очереди, сложность, ограничения**

Алгоритм Дейкстры на основе приоритетной очереди (Dijkstra's algorithm with priority queue) является более эффективным, чем алгоритм на основе сортировки, так как он использует более оптимальную структуру данных для хранения вершин графа в отсортированном порядке по их расстоянию от начальной вершины.

Алгоритм Дейкстры на основе приоритетной очереди работает следующим образом:

1. Инициализируем расстояния до всех вершин графа как бесконечность, за исключением начальной вершины, расстояние до которой равно 0.

2. Создаем пустую приоритетную очередь и добавляем в нее начальную вершину с расстоянием 0.

3. Пока приоритетная очередь не пуста:

a. Извлекаем из очереди вершину с наименьшим расстоянием.

b. Для каждого соседа этой вершины:

i. Вычисляем новое расстояние от начальной вершины до этого соседа через текущую вершину.

ii. Если новое расстояние меньше текущего расстояния до соседа, то обновляем расстояние и добавляем соседа в приоритетную очередь с новым расстоянием.

4. Когда приоритетная очередь пуста, расстояния до всех вершин графа будут найдены.

Сложность алгоритма Дейкстры на основе приоритетной очереди зависит от реализации приоритетной очереди. Если использовать бинарную кучу (binary heap) в качестве приоритетной очереди, то сложность алгоритма будет O(E log V), где E - количество ребер в графе, V - количество вершин. Если использовать Фибоначчиеву кучу (Fibonacci heap), то сложность будет O(E + V log V), что является более эффективным.

Ограничения алгоритма Дейкстры на основе приоритетной очереди также заключаются в том, что он работает только с неотрицательными весами ребер и не может обрабатывать графы с отрицательными циклами. Если в графе есть отрицательный цикл, то алгоритм может зациклиться и не найти кратчайший путь. Для работы с графами с отрицательными весами ребер можно использовать алгоритм Беллмана-Форда (Bellman-Ford algorithm).

1. **Постановка задачи о минимальном остовном дереве (МОД), примеры применения.**

Задача о минимальном остовном дереве заключается в нахождении подмножества ребер связного неориентированного графа, которые соединяют все вершины графа, образуя дерево, и при этом имеют минимальную сумму весов.

Примеры применения задачи о минимальном остовном дереве включают:

1. Прокладка сетей связи: задача о минимальном остовном дереве может быть использована для определения наименьшего набора кабелей, необходимых для связи всех компьютеров в сети.

2. Дизайн электрических цепей: задача о минимальном остовном дереве может быть использована для определения наименьшего набора проводов, необходимых для соединения всех элементов электрической цепи.

3. Планирование маршрутов: задача о минимальном остовном дереве может быть использована для определения наименьшего набора дорог, необходимых для связи всех городов в регионе.

4. Моделирование молекулярных структур: задача о минимальном остовном дереве может быть использована для определения наименьшего набора связей, необходимых для связи всех атомов в молекуле.

5. Анализ социальных сетей: задача о минимальном остовном дереве может быть использована для определения наименьшего набора связей, необходимых для связи всех людей в социальной сети.

1. **Алгоритм Прима на основе сортировки, сложность, ограничения**

Алгоритм Прима на основе сортировки (Prim's algorithm with sorting) используется для поиска минимального остовного дерева в связном взвешенном неориентированном графе. Он работает следующим образом:

1. Инициализируем множество вершин остовного дерева как пустое, а множество всех вершин графа как непосещенные.

2. Выбираем произвольную вершину из графа и добавляем ее в множество вершин остовного дерева.

3. Для каждой вершины из множества вершин остовного дерева находим все ее соседние вершины, которые еще не принадлежат множеству вершин остовного дерева, и вычисляем их веса ребер.

4. Сортируем полученный список ребер по возрастанию весов.

5. Добавляем в множество вершин остовного дерева вершину, соответствующую ребру с наименьшим весом.

6. Повторяем шаги 3-5 до тех пор, пока все вершины графа не будут принадлежать множеству вершин остовного дерева.

Сложность алгоритма Прима на основе сортировки зависит от реализации сортировки. Если использовать быструю сортировку (quick sort) или сортировку слиянием (merge sort), то сложность будет O(E log E), где E - количество ребер в графе. Если использовать сортировку пузырьком (bubble sort) или сортировку вставками (insertion sort), то сложность будет O(E^2).

Ограничения алгоритма Прима на основе сортировки заключаются в том, что он может работать только с связными графами и не может обрабатывать графы с отрицательными весами ребер. Если в графе есть отрицательные веса ребер, то алгоритм может найти неверное остовное дерево. Для работы с графами с отрицательными весами ребер можно использовать алгоритм Крускала (Kruskal's algorithm) или модифицированный алгоритм Прима, который использует приоритетную очередь вместо сортировки для выбора следующей вершины.

1. **Алгоритм Прима на основе приоритетной очереди, сложность, ограничения**

Алгоритм Прима на основе приоритетной очереди — это алгоритм поиска минимального остовного дерева в связном взвешенном неориентированном графе. Он работает за время O(E log V), где E - количество ребер в графе, а V - количество вершин.

Ограничения алгоритма Прима на основе приоритетной очереди такие же, как и у алгоритма на основе сортировки. Он может работать только с связными графами и не может обрабатывать графы с отрицательными весами ребер.

Алгоритм Прима на основе приоритетной очереди работает следующим образом:

1. Инициализируем множество вершин остовного дерева как пустое, а множество всех вершин графа как непосещенные.

2. Выбираем произвольную вершину из графа и добавляем ее в множество вершин остовного дерева.

3. Для каждой вершины из множества вершин остовного дерева находим все ее соседние вершины, которые еще не принадлежат множеству вершин остовного дерева, и вычисляем их веса ребер.

4. Добавляем все полученные ребра в приоритетную очередь, где приоритет определяется весом ребра.

5. Извлекаем из приоритетной очереди ребро с наименьшим весом и добавляем соответствующую ему вершину в множество вершин остовного дерева.

6. Повторяем шаги 3-5 до тех пор, пока все вершины графа не будут принадлежать множеству вершин остовного дерева.

Сложность алгоритма Прима на основе приоритетной очереди составляет O(E log V), где E - количество ребер в графе, а V - количество вершин. Это происходит потому, что каждое ребро добавляется в приоритетную очередь только один раз, а извлечение из очереди занимает O(log V) времени.

Однако, алгоритм Прима на основе приоритетной очереди не может работать с графами с отрицательными весами ребер. Для работы с такими графами можно использовать алгоритм Крускала или модифицированный алгоритм Прима, который использует отрицательные веса ребер для определения приоритетов в приоритетной очереди.

1. **Алгоритм Крускала, сложность, ограничения**

Алгоритм Крускала - это алгоритм поиска минимального остовного дерева в связном взвешенном неориентированном графе. Он работает за время O(E log V), где E - количество ребер в графе, а V - количество вершин.

Ограничения алгоритма Крускала такие же, как и у алгоритма Прима на основе приоритетной очереди. Он может работать только с связными графами и не может обрабатывать графы с отрицательными весами ребер.

Алгоритм Крускала работает следующим образом:

1. Инициализируем множество ребер остовного дерева как пустое, а множество всех вершин графа как непосещенные.

2. Сортируем все ребра графа по весу.

3. Проходимся по отсортированным ребрам и для каждого ребра проверяем, не образует ли оно цикл с уже добавленными ребрами в множество остовного дерева. Если не образует, то добавляем его в множество ребер остовного дерева.

4. Повторяем шаг 3 до тех пор, пока все вершины графа не будут принадлежать множеству вершин остовного дерева.

Сложность алгоритма Крускала составляет O(E log V), где E - количество ребер в графе, а V - количество вершин. Это происходит потому, что сортировка ребер занимает O(E log E) времени, а проверка на циклы с помощью структуры данных Union-Find занимает O(log V) времени.

1. **Сортировка, назначение, основные виды и их особенности**

**Алгоритм сортировки** — это алгоритм для упорядочивания элементов в массиве. В случае, когда элемент в массиве имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

**Сортировка пузырьком / Bubble sort**

Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован. Заметим, что после первой итерации самый большой элемент будет находиться в конце массива, на правильном месте. После двух итераций на правильном месте будут стоять два наибольших элемента, и так далее. Очевидно, не более чем после n итераций массив будет отсортирован. Таким образом, асимптотика в худшем и среднем случае – O(n2), в лучшем случае – O(n).

**Шейкерная сортировка / Shaker sort**

(также известна как сортировка перемешиванием и коктейльная сортировка). Заметим, что сортировка пузырьком работает медленно на тестах, в которых маленькие элементы стоят в конце (их еще называют «черепахами»). Такой элемент на каждом шаге алгоритма будет сдвигаться всего на одну позицию влево. Поэтому будем идти не только слева направо, но и справа налево. Будем поддерживать два указателя begin и end, обозначающих, какой отрезок массива еще не отсортирован. На очередной итерации при достижении end вычитаем из него единицу и движемся справа налево, аналогично, при достижении begin прибавляем единицу и двигаемся слева направо. Асимптотика у алгоритма такая же, как и у сортировки пузырьком, однако реальное время работы лучше.

**Сортировка вставками / Insertion sort**

Создадим массив, в котором после завершения алгоритма будет лежать ответ. Будем поочередно вставлять элементы из исходного массива так, чтобы элементы в массиве-ответе всегда были отсортированы. Асимптотика в среднем и худшем случае – O(n2), в лучшем – O(n). Реализовывать алгоритм удобнее по-другому (создавать новый массив и реально что-то вставлять в него относительно сложно): просто сделаем так, чтобы отсортирован был некоторый префикс исходного массива, вместо вставки будем менять текущий элемент с предыдущим, пока они стоят в неправильном порядке.

**Сортировка Шелла / Shellsort**

Используем ту же идею, что и сортировка с расческой, и применим к сортировке вставками. Зафиксируем некоторое расстояние. Тогда элементы массива разобьются на классы – в один класс попадают элементы, расстояние между которыми кратно зафиксированному расстоянию. Отсортируем сортировкой вставками каждый класс. В отличие от сортировки расческой, неизвестен оптимальный набор расстояний. Существует довольно много последовательностей с разными оценками. Последовательность Шелла – первый элемент равен длине массива, каждый следующий вдвое меньше предыдущего. Асимптотика в худшем случае – O(n2). Последовательность Хиббарда – 2n — 1, асимптотика в худшем случае – O(n1,5), последовательность Седжвика (формула нетривиальна, можете ее посмотреть по ссылке ниже) — O(n4/3), Пратта (все произведения степеней двойки и тройки) — O(nlog2n). Отмечу, что все эти последовательности нужно рассчитать только до размера массива и запускать от большего от меньшему (иначе получится просто сортировка вставками). Также я провел дополнительное исследование и протестировал разные последовательности вида si = a \* si — 1 + k \* si — 1 (отчасти это было навеяно эмпирической последовательностью Циура – одной из лучших последовательностей расстояний для небольшого количества элементов). Наилучшими оказались последовательности с коэффициентами a = 3, k = 1/3; a = 4, k = 1/4 и a = 4, k = -1/5.

**Сортировка выбором / Selection sort**

На очередной итерации будем находить минимум в массиве после текущего элемента и менять его с ним, если надо. Таким образом, после i-ой итерации первые i элементов будут стоять на своих местах. Асимптотика: O(n2) в лучшем, среднем и худшем случае. Нужно отметить, что эту сортировку можно реализовать двумя способами – сохраняя минимум и его индекс или просто переставляя текущий элемент с рассматриваемым, если они стоят в неправильном порядке. Первый способ оказался немного быстрее, поэтому он и реализован.

**Пирамидальная сортировка / Heapsort**

Развитие идеи сортировки выбором. Воспользуемся структурой данных «куча» (или «пирамида», откуда и название алгоритма). Она позволяет получать минимум за O(1), добавляя элементы и извлекая минимум за O(logn). Таким образом, асимптотика O(nlogn) в худшем, среднем и лучшем случае. Реализовывал кучу я сам, хотя в С++ и есть контейнер priority\_queue, поскольку этот контейнер довольно медленный.

**Быстрая сортировка / Quicksort**

Выберем некоторый опорный элемент. После этого перекинем все элементы, меньшие его, налево, а большие – направо. Рекурсивно вызовемся от каждой из частей. В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного. Асимптотика: O(nlogn) в среднем и лучшем случае, O(n2). Наихудшая оценка достигается при неудачном выборе опорного элемента. Моя реализация этого алгоритма совершенно стандартна, идем одновременно слева и справа, находим пару элементов, таких, что левый элемент больше опорного, а правый меньше, и меняем их местами. Помимо чистой быстрой сортировки, участвовала в сравнении и сортировка, переходящая при малом количестве элементов на сортировку вставками. Константа подобрана тестированием, а сортировка вставками — наилучшая сортировка, подходящая для этой задачи (хотя не стоит из-за этого думать, что она самая быстрая из квадратичных).

1. **Алгоритм Шелла, оценки сложности, устойчивость**

**Сортировка Шелла** — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами.

При сортировке Шелла сначала сравниваются и сортируются между собой значения, отстоящие один от другого на некотором расстоянии d. После этого процедура повторяется для некоторых меньших значений d, а завершается сортировка Шелла упорядочиванием элементов при d = 1 (то есть, обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).



**Устойчивость** – это один из параметров трудоемкости алгоритма, который характеризует то, что сортировка не меняет взаимного расположения равных элементов. Сортировка Шелла теряет устойчивость.

Реализация на Java:

**Сложность алгоритма: O(n log2 n).**

for (int step = n / 2; step > 0; step /= 2) {

for (int i = step; i < n; i++) {

for (int j = i - step; j >= 0 && a[j] > a[j + step] ; j -= step) {

int x = a[j];

a[j] = a[j + step];

a[j + step] = x;

}

}

}

1. **Шейкер-сортировка, оценки сложности, устойчивость**

**Шейкер-сортировка** является усовершенствованным методом пузырьковой сортировки.

Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства:

* если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, ее можно исключить из рассмотрения.
* при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две *идеи приводят к модификациям* в методе пузырьковой сортировки.

* От последней перестановки до конца (начала) массива находятся отсортированные элементы. Учитывая данный факт, просмотр осуществляется не до конца (начала) массива, а до конкретной позиции. Границы сортируемой части массива сдвигаются на 1 позицию на каждой итерации.
* Массив просматривается поочередно справа налево и слева направо.
* Просмотр массива осуществляется до тех пор, пока все элементы не встанут в порядке возрастания (убывания).
* Количество просмотров элементов массива определяется моментом упорядочивания его элементов.

Следующая таблица отражает временную сложность алгоритма шейкерной сортировки для трех случаев.

|  |  |  |
| --- | --- | --- |
| Лучший случай | Средний случай | Наихудший случай |
| O(n) | O(n2) | O(n2) |

Устойчивость – это параметр, который отвечает за то, что сортировка не меняет взаимного расположения равных элементов;

Шейкерная сортировка является устойчивой.

Рассмотрим алгоритм Шейкер-сортировки на примере. Дана последовательность

44 55 12 42 94 18 6 67

**-> -> -> -> ->**

44 12 42 **55** 18 6 67 **94**

**<- <- <- <- <-**

**6**  44 12 42 55 18 67 **94**

**-> -> ->**

**6**  12 42 **44** 18 **55** **67** **94**

**<- <-**

**6**  **12** **18** 42 44 55 **67** **94**

**перемещений нет**

**6**  **12** **18** 42 44 **55** **67** **94**

**6**  **12** **18** **42** **44** **55** **67** **94**

**Реализация алгоритма Шейкер-сортировки на C#**

Каждое повторение цикла while() представляет собой шаг сортировки.

**void** Swap(**int** \*Mas, **int** i) //функция обмена

{

**int** temp;

temp=Mas[i];

Mas[i]=Mas[i-1];

Mas[i-1]=temp;

}

**void** ShakerSort(**int** \*Mas, **int** Start, **int** N) //функция шейкерной сортировки

{

**int** Left, Right, i;

Left=Start;

Right=N-1;

**while** (Left<=Right)

{

**for** (i=Right; i>=Left; i--)

**if** (Mas[i-1]>Mas[i]) Swap(Mas, i);

Left++;

**for** (i=Left; i<=Right; i++)

**if** (Mas[i-1]>Mas[i]) Swap(Mas, i);

Right--;

}

}

1. **Сортировка Хоара, оценки сложности, устойчивость**

Этот алгоритм, чаще называемый просто «быстрая сортировка», придуман английским ученым Чарльзом Хоаром в 1960 году и использует стратегию **«разделяй и властвуй»**.

Шаги алгоритма таковы:

1. Выбираем в массиве некоторый элемент, который будем называть *опорным элементом*.

2. Операция разделения массива: реорганизуем массив таким образом, чтобы все элементы, меньшие или равные опорному элементу, оказались слева от него, а все элементы, большие опорного — справа от него.

**Обычный алгоритм этой операции:**

* Два индекса — l и r, приравниваются к минимальному и максимальному индексу разделяемого массива соответственно.
* Вычисляется индекс опорного элемента m.
* Индекс l последовательно увеличивается до m до тех пор, пока l-й элемент не превысит опорный.
* Индекс r последовательно уменьшается до m до тех пор, пока r-й элемент не окажется меньше либо равен опорному.
* Если r = l — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент.
* Если l < r — найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений l и r, которые были достигнуты. Следует учесть, что если какая-либо граница (l или r) дошла до опорного элемента, то при обмене значение m изменяется на r-й или l-й элемент соответственно.

**Асимптотика алгоритма**

Сложность алгоритма быстрой сортировки Хоара зависит от метода выбора барьерного элемента. В лучшем случае при каждом выборе барьерного элемента должен выбираться медианный элемент массива. Но поиск медианного элемента — сложная задача, её нельзя решить быстро. Если выбрать первый элемент фрагмента списка A[l] или последний A[r], то если список A уже упорядочен, сложность алгоритма будет **O(n2)**, так как на каждом рекурсивном вызове от большей части списка будет отделяться всего один элемент.

Поэтому в алгоритме быстрой сортировки Хоара, как правило, в качестве барьерного элемента выбирается случайный элемент списка. Тогда алгоритм становится вероятностным — время его работы зависит от того, каким будет случайно выбранный элемент. Возможна (но крайне маловероятна) ситуация, когда всегда будет выбираться наименьший элемент, и в этом случае алгоритм будет работать за **O(n2)**.

В теории вероятностей доказывается, чти при случайном выборе элемента списка и разбиении его на две части, размер большей из двух получившихся частей будет в среднем равен **3n/4**. В этом случае глубина рекурсии в среднем будет составлять порядка **log n**, а средняя сложность алгоритма быстрой сортировки Хоара — **O(n log n)**.

Стоит отметить, что быстрая сортировка может оказаться малоэффективной на массивах, состоящих из небольшого числа элементов, поэтому при работе с ними разумнее отказаться от данного метода. В целом алгоритм неустойчив, а также использование рекурсии в неверно составленном коде может привести к переполнению стека. Но, несмотря на эти и некоторые другие минусы, быстрая сортировка все же является одним из самых эффективных и часто используемых методов.

1. **Пирамидальная сортировка, оценки сложности, устойчивость**

**Пирамидальная сортировка** — алгоритм сортировки, работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за  𝑂(𝑙𝑜𝑔 𝑛) операций при сортировке элементов. Количество применяемой служебной памяти не зависит от размера массива (то есть, 𝑂(1)).

Сортировка пирамидой использует бинарное сортирующее дерево.

**Сортирующее дерево — это такое дерево, у которого выполнены условия:**

1.Каждый лист имеет глубину либо d, либо d-1, d— максимальная глубина дерева.

2.Значение в любой вершине не меньше (другой вариант — не больше) значения её потомков.

Построение пирамиды занимает O(n log n) операций, причем более точная оценка дает даже O(n) за счет того, что реальное время выполнения downheap зависит от высоты уже созданной части пирамиды.

Вторая фаза занимает O(n log n) времени: O(n) раз берется максимум и происходит просеивание бывшего последнего элемента. Плюсом является стабильность метода: среднее число пересылок (n log n)/2, и отклонения от этого значения сравнительно малы.

Пирамидальная сортировка не использует дополнительной памяти.

Метод не является устойчивым: по ходу работы массив так "перетряхивается", что исходный порядок элементов может измениться случайным образом.

Поведение неестественно: частичная упорядоченность массива никак не учитывается.

1. **Сортировка подсчетом, оценки сложности, устойчивость**

Сортировка подсчётом – сортировка посредством подсчёта англ. sorting by counting) — алгоритм сортировки, в котором используется диапазон чисел сортируемого массива (списка) для подсчёта совпадающих элементов. Применение сортировки подсчётом целесообразно лишь тогда, когда сортируемые числа имеют (или их можно отобразить в) диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством, например, миллион натуральных чисел меньших 1000.

**Простой алгоритм**

Это простейший вариант алгоритма. Создать вспомогательный массив , состоящий из нулей, затем последовательно прочитать элементы входного массива A, для каждого A[i] увеличить на единицу. Теперь достаточно пройти по массиву C, для каждого в массив A последовательно записать число раз.

**SimpleCountingSort:**

for i = 0 to k - 1

C[i] = 0;

for i = 0 to n - 1

C[A[i]] = C[A[i]] + 1;

b = 0;

for j = 0 to k - 1

for i = 0 to C[j] - 1

A[b] = j;

b = b + 1;

**Устойчивый алгоритм**

В этом варианте помимо входного массива потребуется два вспомогательных массива — для счётчика и для отсортированного массива. начала следует заполнить массив C нулями, и для каждого увеличить на 1. Далее подсчитывается количество элементов меньших или равных . Для этого каждый C[j], начиная с C[1], увеличивают на C[j - 1]. Таким образом в последней ячейке будет находиться количество элементов от до существующих во входном массиве. На последнем шаге алгоритма читается входной массив с конца, значение уменьшается на 1 и в каждый записывается . Алгоритм устойчив.

**StableCountingSort**

for i = 0 to k - 1

C[i] = 0;

for i = 0 to n - 1

C[A[i]] = C[A[i]] + 1;

for j = 1 to k - 1

C[j] = C[j] + C[j - 1];

for i = n - 1 to 0

C[A[i]] = C[A[i]] - 1;

B[C[A[i]]] = A[i];

**Анализ**

В первом алгоритме первые два цикла работают за O(k) и O(n), соответственно; двойной цикл за O(n+k). Во втором алгоритме циклы занимают O(k), O(n), O(k) и O(n), соответственно. Итого два алгоритма имеют линейную временную трудоёмкость O(n+k). Используемая память в первых двух алгоритмах равна O(k), а в третьем O(n + k).

1. **Волновой алгоритм,пошаговое описание,сложность**

Волновой алгоритм — это алгоритм поиска кратчайшего пути в графе без весовых рёбер, который основан на распространении волны из начальной вершины до конечной.

Пошаговое описание алгоритма:

1. Инициализируем начальную вершину, помещаем ее в очередь или стек.
2. Пока очередь или стек не пусты, выполняем следующие действия:

a) Извлекаем вершину из очереди или стека.

б) Для каждой смежной вершины, которая еще не была посещена, устанавливаем ее метку равной метке текущей вершины плюс 1 и помещаем ее в очередь или стек для дальнейшей обработки.

1. Когда мы достигаем конечной вершины или когда очередь или стек пусты, алгоритм завершается.

Волновой алгоритм имеет сложность O(V+E), где V - число вершин в графе, а E - число ребер. Это связано с тем, что каждая вершина и каждое ребро посещаются только один раз. Однако, если граф имеет очень большое число вершин и ребер, то время работы алгоритма может быть значительным.

Волновой алгоритм может быть использован для поиска кратчайшего пути в графе с невзвешенными ребрами, но не может быть использован для графов с отрицательными весами, так как это может привести к зацикливанию и неправильному результату. Для графов с отрицательными весами обычно используются алгоритмы, такие как алгоритм Беллмана-Форда или алгоритм Дейкстры с учетом отрицательных весов ребер.

Пошаговое описание алгоритма и его сложность я уже описал в предыдущем ответе. Приведу пример реализации на Python:

def wave\_algorithm(graph, start, end):

queue = [start]

visited = {start: 0}

while queue:

node = queue.pop(0)

if node == end:

return visited[node]

for neighbor in graph[node]:

if neighbor not in visited:

visited[neighbor] = visited[node] + 1

queue.append(neighbor)

return -1 # если путь не найден

# пример использования

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

print(wave\_algorithm(graph, 'A', 'F')) # 2

Здесь `graph` — это представление графа в виде словаря, где ключи — это вершины графа, а значения - списки смежных вершин. `start` и `end` — это начальная и конечная вершины соответственно. Алгоритм начинает с инициализации начальной вершины `start` и ее расстояния от начальной вершины равным 0. Затем алгоритм продолжает работу, пока очередь `queue` не пуста. На каждой итерации он извлекает вершину из очереди `queue` и для каждой ее смежной вершины, которая еще не была посещена, устанавливает ее метку равной метке текущей вершины плюс 1 и помещает ее в очередь `queue` для дальнейшей обработки. Когда алгоритм достигает конечной вершины `end` или когда очередь `queue` пуста, он завершает работу и возвращает расстояние от начальной вершины до конечной вершины в `visited[end]`. Если путь не найден, то возвращается -1.

Сложность алгоритма в данном случае составляет O(V+E), где V — это количество вершин в графе, а E - количество ребер. В реализации применяется поиск в ширину (BFS), который обходит каждую вершину и каждое ребро только один раз.

**34. Алгоритм Флойда-Уоршалла, пошаговое описание, сложность**

Алгоритм Флойда-Уоршалла — это алгоритм поиска кратчайших путей между всеми парами вершин в ориентированном или неориентированном графе с возможными отрицательными весами ребер. Он основывается на построении матрицы расстояний между парами вершин.

Пошаговое описание алгоритма:

1. Инициализируем матрицу расстояний D размера n x n, где n — это количество вершин в графе. Если между вершинами i и j есть ребро, то D[i][j] присваивается вес этого ребра. Если между вершинами i и j нет ребра, то D[i][j] присваивается бесконечность.
2. Для каждой вершины k от 1 до n выполняем следующие действия:

a) Для каждой пары вершин i и j от 1 до n проверяем, можно ли улучшить расстояние от i до j, используя вершину k как промежуточную. Для этого сравниваем текущее расстояние D[i][j] с суммой расстояний D[i][k] и D[k][j]. Если сумма расстояний меньше текущего расстояния, то обновляем значение D[i][j].

1. По завершении алгоритма матрица D будет содержать кратчайшие расстояния между всеми парами вершин.

Сложность алгоритма Флойда-Уоршалла составляет O(n^3), где n - количество вершин в графе. Это связано с тем, что для каждой вершины k выполняется проверка расстояний между всеми парами вершин.

Пример реализации на Python:

def floyd\_warshall(graph):

n = len(graph)

dist = [[float('inf') for \_ in range(n)] for \_ in range(n)]

for i in range(n):

for j in range(n):

if i == j:

dist[i][j] = 0

elif j in graph[i]:

dist[i][j] = graph[i][j]

for k in range(n):

for i in range(n):

for j in range(n):

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

return dist

# пример использования

graph = {

0: {1: 2, 2: 6, 3: 4},

1: {4: 3},

2: {3: 1, 4: 7},

3: {4: 2},

4: {}

}

print(floyd\_warshall(graph))

Здесь graph — это представление графа в виде словаря, где ключи - это вершины графа, а значения - это словари, где ключи - это смежные вершины, а значения - это веса ребер. Алгоритм начинается с инициализации матрицы расстояний dist, гдекаждый элемент dist[i][j] равен весу ребра из вершины i в вершину j. Если между вершинами i и j нет ребра, то значение равно бесконечности. Затем для каждой вершины k от 1 до n алгоритм проверяет, можно ли улучшить расстояние между каждой парой вершин i и j, используя вершину k как промежуточную. Если сумма расстояний dist[i][k] и dist[k][j] меньше текущего значения dist[i][j], то обновляем значение dist[i][j]. По завершении алгоритма функция возвращает матрицу расстояний dist.

Сложность алгоритма в данном случае составляет O(n^3), где n - количество вершин в графе. В реализации используется три вложенных цикла, которые просматривают каждую вершину и каждое ребро только один раз.