Алгоритм Евклида для нахождения НОД двух целых чисел a, b

Рекурсивная версия

```
def gcd_rek(a, b):
    if b == 0:
        return a
    return gcd_rek(b, a % b)
```

Итеративная версия

```
def gcd_itr(a, b):
    while b != 0:
        t = b
        b = a % b
        a = t
    return a
```

Наименьшее общее кратное

```
def lcm(n, m):
    return n // gcd_rek(n, m) * m
# НОД нескольких чисел (список входное)
def gcd_list(numbers):
    result = 0
    for item in numbers:
        result = gcd_rek(result, item)
    return result
```

НОД нескольких чисел (список входное)

```
def gcd_list(numbers):
    result = 0
    for item in numbers:
        result = gcd_rek(result, item)
    return result
```

Максимальный НОД списка чисел, который можно получить из префиксов и суффиксов массива

```
def gcd_array(numbers):
    n = len(numbers)
    suffix = [0] * (n + 1)
    for i in range(n - 1, -1, -1):
        suffix[i] = gcd_rek(suffix[i + 1], numbers[i])
    answer, prefix = 0, 0
    for j in range(n):
        answer = max(answer, gcd_rek(prefix, suffix[j + 1]))
        prefix = gcd_rek(prefix, numbers[j])
    return answer
```

Функция для объединения отрезков

```
def merge_segments(segments):
    res = []
    for segm in segments:
        if not res or segm[0] > res[-1][1]:
            res.append(segm)
        else:
            res[-1] = (res[-1][0], max(res[-1][1], segm[1]))
    return res
```

Обход в глубину DFS (Смежность вершин)

```
inc = {
    1: [2, 8],
    2: [1, 3, 8],
    3: [2, 4, 8],
    4: [3, 7, 9],
    5: [6, 7],
    6: [5],
    7: [4, 5, 8],
    8: [1, 2, 3, 7],
    9: [4],
}
visited = set()
def dfs(v):
    if v in visited:  # Если вершина уже посещена, выходим
        return
    visited.add(v)  # Посетили вершину v
    for i in inc[v]:  # Все смежные с v вершины
        if not i in visited:
            dfs(i)
```

Поиск в ширину BFS (Смежность вершин)

```
inc = {
    1: [2, 8],
    2: [1, 3, 8],
    3: [2, 4, 8],
    4: [3, 7, 9],
    5: [6, 7],
    6: [5],
    7: [4, 5, 8],
    8: [1, 2, 3, 7],
    9: [4],
}
visited = set()  # Посещена ли вершина?
Q = []  # Очередь
BFS = []
def bfs(v):
    if v in visited:  # Если вершина уже посещена, выходим
        return
    visited.add(v)  # Посетили вершину v
    BFS.append(v)  # Запоминаем порядок обхода
    # print("v = %d" % v)
    for i in inc[v]:  # Все смежные с v вершины
        if not i in visited:
            Q.append(i)
    while Q:
        bfs(Q.pop(0))
```

Бинарный поиск в отсортированном списке

```
def binary_search(arr, x):
    left, right = 0, len(arr) - 1
    while left <= right:
```

```
        mid = left + (right - left) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Быстрое возведение в степень с модулем

```
def power(a, b, mod):
    res = 1
    a = a % mod
    while b > 0:
        if b & 1:
            res = (res * a) % mod
        a = (a * a) % mod
        b >>= 1
    return res
```

Генерирует список простых чисел до n с помощью решета Эстафена

```
def sieve(n):
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(n ** 0.5) + 1):
        if is_prime[i]:
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
    return is_prime
```

Префиксные суммы для заданного массива

```
def prefix_sum(arr):
    psum = [0] * (len(arr) + 1)
    for i in range(len(arr)):
        psum[i + 1] = psum[i] + arr[i]
    return psum
```

Работа со словарями

clear() - удаляет все элементы словаря
copy() - создает копию словаря
get() возвращает значение по ключу
del <имя словаря>[key] удаление элемента с ключом
pop(key) удаление по клюу
popitem() удаляет последний элемент из словаря
clear() очистить словарь
dict(sorted(<словарь>.item(), key=lambda item: item[1])) сортировка словаря по значениям

Работа со строками

chr() преобразует целое число в символ
ord() преобразует символ в целое число
.capitalize() приводит первую букву в верхний регистр, остальные в нижний
.lower() в нижний регистр
.swapspace() меняет регистр букв на противоположный
.title() первые буквы всех слов в заглавные

.upper() все буквы в верхний регистр
count(<строка>) посчитать кол-во вхождений
.endswith(<строка>) проверка что строка окнчивается на под строку,
startwith - начинается с под строки
find() найти индекс первого вхождения в стркоу
.isalnum() проверка что строка только из букв и цифр
.isalpha() только из букв
.isdigit() строка = число целое
.islower() проверка, что все цифры в нижнем регистре
.isupper() все в верхнем регистре
.istitle() начинаются с заглавной буквы

БИНАРНЫЙ ПОИСК (поиск в отсортированном массиве):

```
int binary_search(std::vector<int>& array, int num) {
	int low = 0; // переменная индекса начала исследуемого отрезка
	int high = array.size() - 1; // переменная индекса конца исследуемого отрезка
	int mid; // переменная для хранения середины исследуемого отрезка
	while (low <= high) { // пока эта часть не сократится до одного элемента
		mid = (low + high) / 2; // каждый раз делим исследуемую область пополам
		if (array[mid] == num) { // нашли что искали
			return mid;
		}
		else if (array[mid] > num) { // много взяли
			high = mid - 1;
		}
		else { // мало взяли
			low = mid + 1;
		}
	}
	return -1;
}
```

БЫСТРАЯ СОРТИРОВКА:

Алгоритм быстрой сортировки работает так: сначала в массиве выбирается элемент, который называется опорным. Теперь мы находим элементы меньше опорного и элементы больше опорного.

Этот процесс называется *разделением*. Теперь у вас имеются:

- Подмассив элементов меньше опорного;
- опорный элемент;
- подмассив элементов больше опорного.

Два подмассива не отсортированы — они просто выделены из исходного массива. Но если бы они были отсортированы, то провести сортировку всего массива было бы несложно.

Если бы подмассивы были отсортированы, то их можно было бы объединить в порядке “левый помассив — опорный элемент — правый подмассив” и получить отсортированный массив.

Как отсортировать подмассивы? Базовый случай быстрой сортировки уже знает, как сортировать подмассивы из двух элементов (левый подмассив) и пустые массивы (правый подмассив). Следовательно, если применить алгорит быстрой сортировки к двум подмассивам, а затем объеденить результаты, получится отсортированный массив!

Пример кода быстрой сортировки:

```
int partition(std::vector<int>& array, int low, int high) { // разделение
// (поиск индекса опорного элемена)
    int pivot = array[high]; // опорный элемент
    int i = low - 1; // элемент который будет меняться по порядку
// начиная с начала массива
    for (int j = low; j < high; ++j) {
        if (array[j] <= pivot) { // Если текущий элемент меньше опорного,
                    // то меняем местами
            i++;
            std::swap(array[i], array[j]);
        }
    }
    std::swap(array[i+1], array[high]);
    return i + 1;
}
void quickSort(std::vector<int>& array, int low, int high) {
    if (low < high) {
        int indexPart = partition(array, low, high);

        quickSort(array, low, indexPart - 1); // Сортируем левую часть
        quickSort(array, indexPart + 1, high); // Сортируем правую часть
    }
}
```

Данный программный код выполняется по схеме Ломуто. В схеме Ломуто опорный элемент изначально выбирается последним в массиве. Затем инициализируется индекс i для перемещения элементов. После начинается цикл программы, в котором j является индексом по перебору элементов в сортируемой части массива: если текущий элемент меньше опорного, то нужно произвести перестановку, по итогу перестановки элемент под индексом i будет являться элементом до опорного элемента. После основного цикла остаётся только вернуть опорный элемент на свою законную позицию — туда где элементы до него будут гарантировано меньше, а после — гарантировано больше. Дальше уже пляшем от найденного опорного элемента, который будет являться то началом сортируемой области, если мы сортируем правую часть, то концом сортируемой области, если мы сортируем левую часть.

ПОИСК В ШИРИНУ:

это алгоритм, который работает на графах. Графы в свою очередь являются абстрактными типами данных.

Алгоритм поиска в ширину решает в свою очередь два типа задач:

- Существует ли путь от узла A к узлу B?
- Какой кратчайший путь от узла A до узла B?

Проверять связи в графе нужно в порядке их добавления, потому что иначе для задачи поиска кратчайшего пути будет найдена не кратчайшая дорога. Для операций такого рода существует специальная структура данных, которая называется *очередью*.

Пример кода поиска в ширину:

```
class Graph {
private:
    int V; // количество вершин
    std::vector<std::vector<int>> adj; // список смежностей
public:
    Graph(int V) : V(V) {
        adj.resize(V);
    }
```

```
    // добавляем ребро между вершинами u и v
    void addEdge(int u, int v) {
        adj[u].push_back(u);
    }
    // алгоритм поиска в ширину
    void BFS(int start) {
        std::vector<bool> visited(V, false); // вектор посещённых
вершин
        std::queue<int> q; // очередь для поочередной проверки узлов
        q.push(start); // добавляем начальный узел
        visited[start] = true; // помечаем его пройденным
        while (!q.empty()) { // пока очередь не пуста продолжаем обход
графа
            int curr = q.front(); // текущий элемент на очереди
            q.pop(); // убираем его из очереди
            for (int node : adj[curr]) { // проверяем все смежные
вершины по отношению к текущей
                if (!visited[node]) { // если смежная вершина не была
проверена, то
                    q.push(node); // добавляем её в очередь
                    visited[node] = true; // и помечаем посещённой
                }
            }
        }
    }
};
```

АЛГОРИТМ ДЕЙКСТРЫ:

используется для нахождения кратчайших путей от одной вершины до всех остальных в графе с неотрицательными весами рёбер. Это жадный алгоритм, который работает, постепенно расширяя область уже найденных кратчайших путей.

Шаги алгоритма:

- Инициализация: Мы задаём расстояние до начальной вершины (источника) равным нулю, а расстояния до всех других вершин — бесконечности (∞).

- Посещение вершин: На каждом шаге выбирается вершина с минимальным расстоянием среди ещё не обработанных вершин. Мы фиксируем её как текущую вершину и рассматриваем всех её соседей.

- Обновление расстояний: Для каждой смежной с текущей вершины вершины, проверяем, даёт ли путь через текущую вершину более короткое расстояние. Если да, то обновляем расстояние для этой смежной вершины.

- Повторение: Повторяем шаги до тех пор, пока не будут обработаны все вершины.

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;
// Структура для хранения рёбер графа
struct Edge {
    int destination;  // Конечная вершина ребра
    int weight;       // Вес ребра
};
// Алгоритм Дейкстры
void dijkstra(int source, const vector<vector<Edge>>& graph) {
```

```
    int n = graph.size();  // Число вершин в графе
    vector<int> distances(n, INT_MAX);  // Массив расстояний,
изначально ∞
    distances[source] = 0;  // Расстояние до начальной вершины — 0
    // Приоритетная очередь для обработки вершин (минимальная куча)
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
    pq.push({0, source});  // Добавляем начальную вершину с
расстоянием 0
    while (!pq.empty()) {
        int dist = pq.top().first;  // Текущее минимальное расстояние
        int u = pq.top().second;    // Вершина с этим расстоянием
        pq.pop();
        // Пропускаем, если текущее расстояние больше уже найденного
        if (dist > distances[u]) continue;
        // Проходим по всем смежным вершинам
        for (const Edge& edge : graph[u]) {
            int v = edge.destination;
            int weight = edge.weight;
            // Проверяем, улучшает ли путь через вершину u расстояние
до v
            if (distances[u] + weight < distances[v]) {
                distances[v] = distances[u] + weight;  // Обновляем
расстояние
                pq.push({distances[v], v});  // Добавляем вершину в
очередь
            }
        }
    }
    // Выводим кратчайшие расстояния от источника
    cout << "Vertex\tDistance from Source\n";
    for (int i = 0; i < n; i++) {
        cout << i << "\t" << distances[i] << "\n";
    }
}
```

ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ:

это метод решения задач, которые могут быть разбиты на перекрывающиеся подзадачи. Основная идея — разбить сложную задачу на более простые подзадачи, решить каждую из них один раз и сохранить результат для использования в будущем, что предотвращает повторные вычисления и значительно ускоряет выполнение программы.

Два основных подхода к динамическому программированию:

- Снизу вверх (итеративно): сначала решаем мелкие подзадачи, а затем на их основе строим решение для всей задачи.

- Сверху вниз (мемоизация): используется рекурсия и запоминаются результаты уже вычисленных подзадач.

Пример задачи: "Задача о рюкзаке" (Knapsack Problem)

Условие задачи:

У нас есть набор предметов, каждый из которых имеет определённый вес и ценность. Цель — выбрать предметы таким образом, чтобы их суммарная ценность была максимальной, но при этом их суммарный вес не превышал допустимого значения W.

Входные данные:

Массив весов weights и массив ценностей values, где weights[i] и values[i] представляют вес и ценность предмета i.

Число W — максимальный допустимый вес.

Решение с использованием динамического программирования:

Создадим двумерный массив dp, где dp[i][w] будет хранить максимальную ценность для первых i предметов при ограничении на вес w.

Рекуррентная формула:

Если вес предмета i больше текущего предельного веса w, то не включаем этот предмет:

$dp[i][w]=dp[i-1][w]$.

Если вес предмета i меньше или равен w, то выбираем максимальное значение между:

- не включать предмет i: $dp[i-1][w]$
- включить предмет i: $dp[i-1][w-weights[i]]+values[i]$

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Функция для решения задачи о рюкзаке с помощью DP
int knapsack(int W, const vector<int>& weights, const vector<int>&
values) {
    int n = weights.size();  // количество предметов
    // Создаем DP-массив размером (n+1) x (W+1), инициализируем нулями
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));
    // Заполняем DP-массив
    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= W; w++) {
            if (weights[i - 1] <= w) {
                // Выбираем максимальное значение между включением и
не включением предмета
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i -
1]] + values[i - 1]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    // Возвращаем максимальную ценность для всех предметов и
ограничения на вес W
    return dp[n][W];
}
int main() {
    int W = 50;  // Максимальный вес
    vector<int> weights = {10, 20, 30};  // Веса предметов
    vector<int> values = {60, 100, 120};  // Ценности предметов
    cout << "Максимальная ценность: " << knapsack(W, weights, values)
<< endl;
    return 0;
}
```

Комбинаторика на Python

С помощью функции permutations можно сгенерировать все перестановки для итерируемого объекта.

```
for i in permutations('abc'):
    print(i, end=' ') # abc acb bac bca cab cba
```

С помощью размещений с повторениями можно легко перебрать все строки фиксированной длины, состоящие из заданных символов

```
for i in product('abc', repeat=2):
    print(i, end=' ') # aa ab ac ba bb bc ca cb cc
```

С помощью сочетаний без повторений можно перебрать все наборы не повторяющихся букв из заданной строки, массива или другого итерируемого объекта без учета порядка

```
for i in combinations('abcd', 2):
    print(i, end=' ') # ab ac ad bc bd cd
```

Результат аналогичен вызову combinations, но в результат также добавлены множества с одинаковыми элементами.

```
for i in combinations_with_replacement('abcd', 2):
    print(i, end=' ') # aa ab ac ad bb bc bd cc cd dd
```