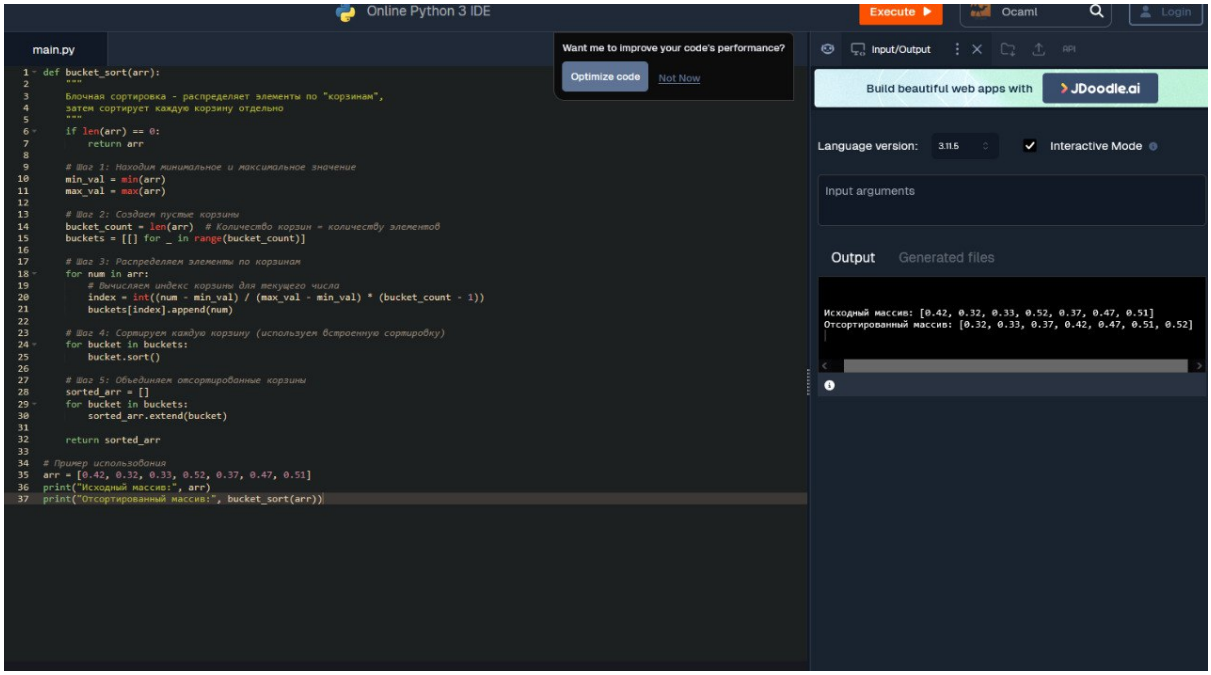
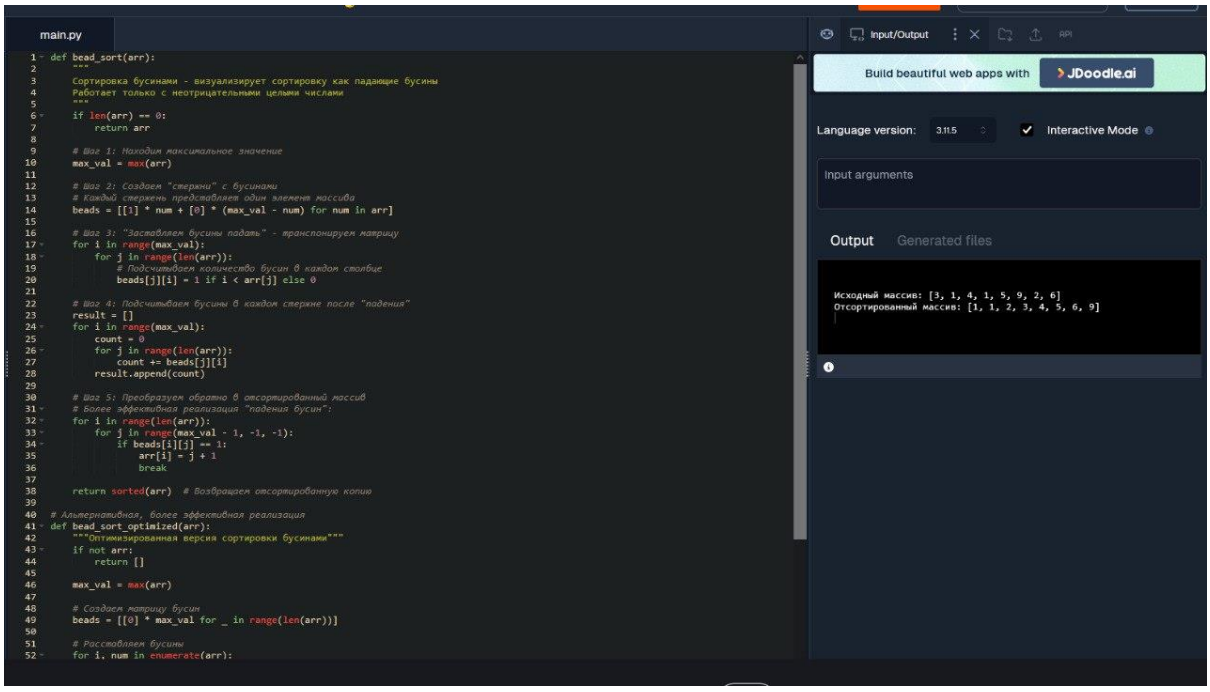
Bucket_sort



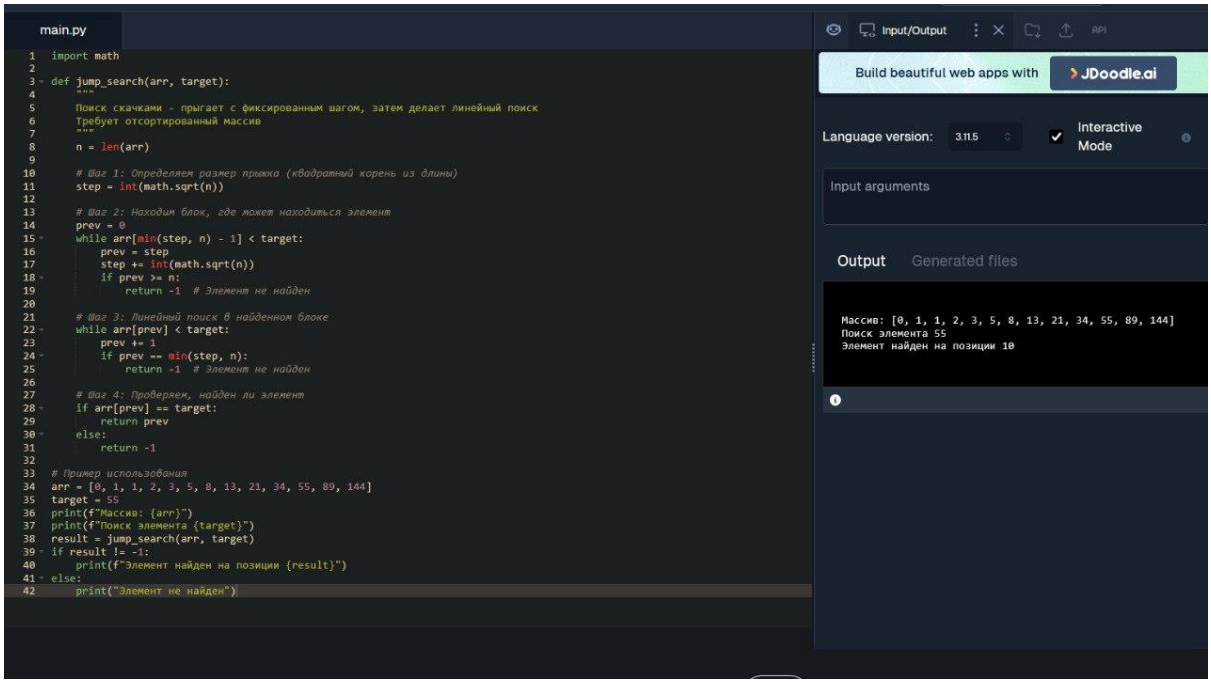

```python
def bucket_sort(arr):
    """
    Блочная сортировка - распределяет элементы по "корзинам",
    затем сортирует каждую корзину отдельно
    """
    if len(arr) == 0:
        return arr

    # Шаг 1: Находим минимальное и максимальное значение
    min_val = min(arr)
    max_val = max(arr)

    # Шаг 2: Создаем пустые корзины
    bucket_count = len(arr)  # Количество корзин = количеству элементов
    buckets = [[] for _ in range(bucket_count)]

    # Шаг 3: Распределяем элементы по корзинам
    for num in arr:
        # Вычисляем индекс корзины для текущего числа
        index = int((num - min_val) / (max_val - min_val) * (bucket_count - 1))
        buckets[index].append(num)

    # Шаг 4: Сортируем каждую корзину (используем встроенную сортировку)
    for bucket in buckets:
        bucket.sort()

    # Шаг 5: Объединяем отсортированные корзины
    sorted_arr = []
    for bucket in buckets:
        sorted_arr.extend(bucket)

    return sorted_arr

# Пример использования
arr = [0.42, 0.32, 0.33, 0.52, 0.37, 0.47, 0.51]
print("Исходный массив:", arr)
print("Отсортированный массив:", bucket_sort(arr))
```

Bead_sort



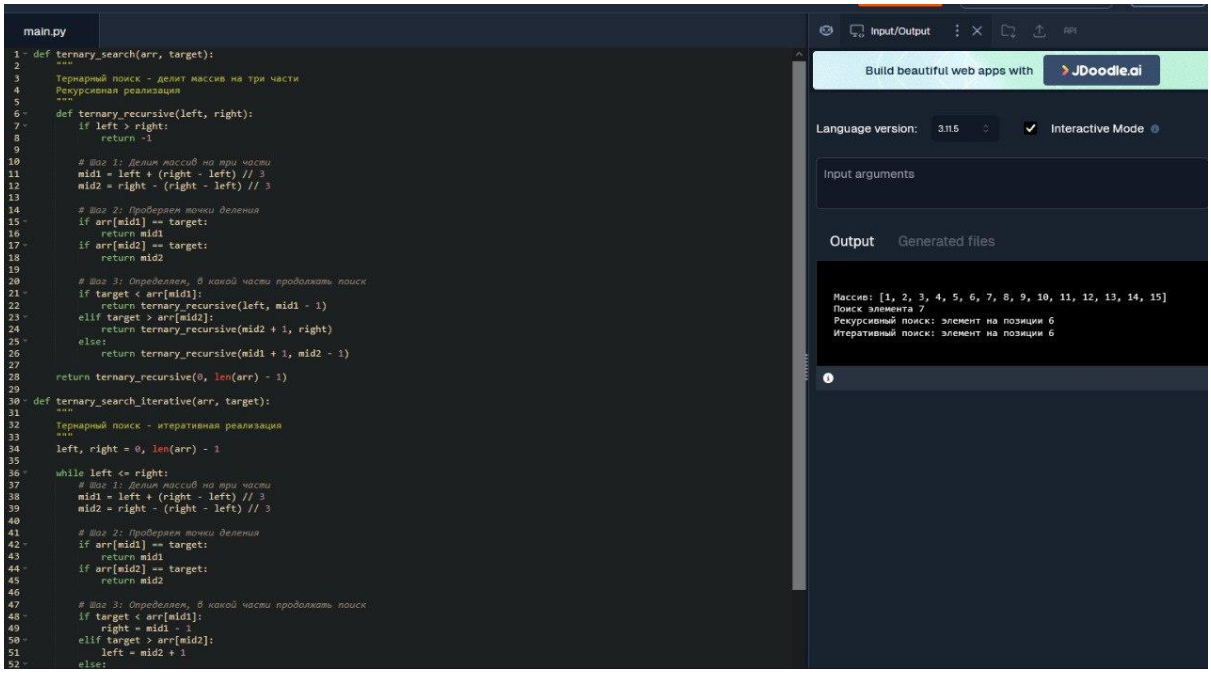

```python
def bead_sort(arr):
    """
    Сортировка бусинами - визуализирует сортировку как падающие бусины
    Работает только с неотрицательными целыми числами
    """
    if len(arr) == 0:
        return arr

    # Шаг 1: Находим максимальное значение
    max_val = max(arr)

    # Шаг 2: Создаем "стержни" с бусинами
    # Каждый стержень представляет один элемент массива
    beads = [[1] * num + [0] * (max_val - num) for num in arr]

    # Шаг 3: "Заставляем бусины падать" - транспонируем матрицу
    for i in range(max_val):
        for j in range(len(arr)):
            # Подсчитываем количество бусин в каждом столбце
            beads[j][i] = 1 if i < arr[j] else 0

    # Шаг 4: Подсчитываем бусины в каждом стержне после "падения"
    result = []
    for i in range(max_val):
        count = 0
        for j in range(len(arr)):
            count += beads[j][i]
        result.append(count)

    # Шаг 5: Преобразуем обратно в отсортированный массив
    # Более эффективная реализация "падения бусин":
    for i in range(len(arr)):
        for j in range(max_val - 1, -1, -1):
            if beads[i][j] == 1:
                arr[i] = j + 1
                break

    return sorted(arr)  # Возвращаем отсортированную копию

# Альтернативная, более эффективная реализация
def bead_sort_optimized(arr):
    """Оптимизированная версия сортировки бусинами"""
    if not arr:
        return []

    max_val = max(arr)

    # Создаем матрицу бусин
    beads = [[0] * max_val for _ in range(len(arr))]

    # Расставляем бусины
    for i, num in enumerate(arr):
```

Jump_search



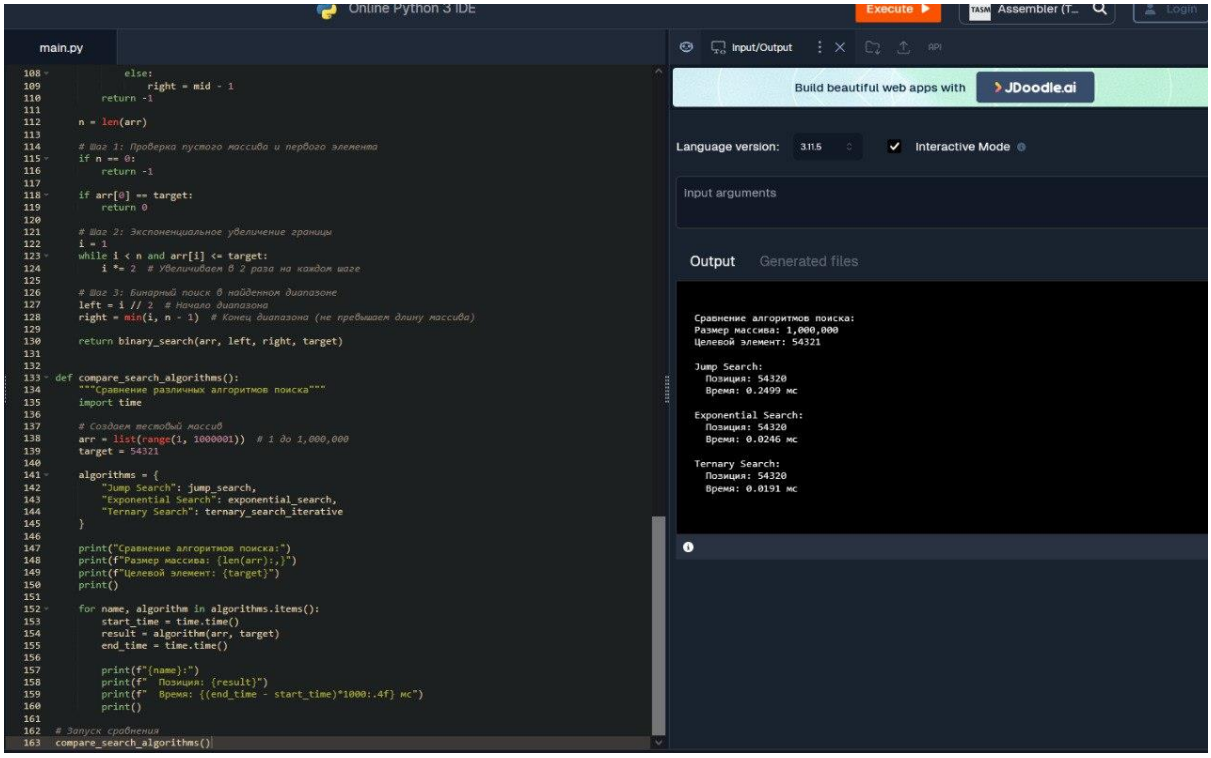

```python
import math

def jump_search(arr, target):
    """
    Поиск скачками - прыгает с фиксированным шагом, затем делает линейный поиск
    Требует отсортированный массив
    """
    n = len(arr)

    # Шаг 1: Определяем размер прыжка (квадратный корень из длины)
    step = int(math.sqrt(n))

    # Шаг 2: Находим блок, где может находиться элемент
    prev = 0
    while arr[min(step, n) - 1] < target:
        prev = step
        step += int(math.sqrt(n))
        if prev >= n:
            return -1  # Элемент не найден

    # Шаг 3: Линейный поиск в найденном блоке
    while arr[prev] < target:
        prev += 1
        if prev == min(step, n):
            return -1  # Элемент не найден

    # Шаг 4: Проверяем, найден ли элемент
    if arr[prev] == target:
        return prev
    else:
        return -1

# Пример использования
arr = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
target = 55
print(f"Массив: {arr}")
print(f"Поиск элемента {target}")
result = jump_search(arr, target)
if result != -1:
    print(f"Элемент найден на позиции {result}")
else:
    print("Элемент не найден")
```

Ternary_search

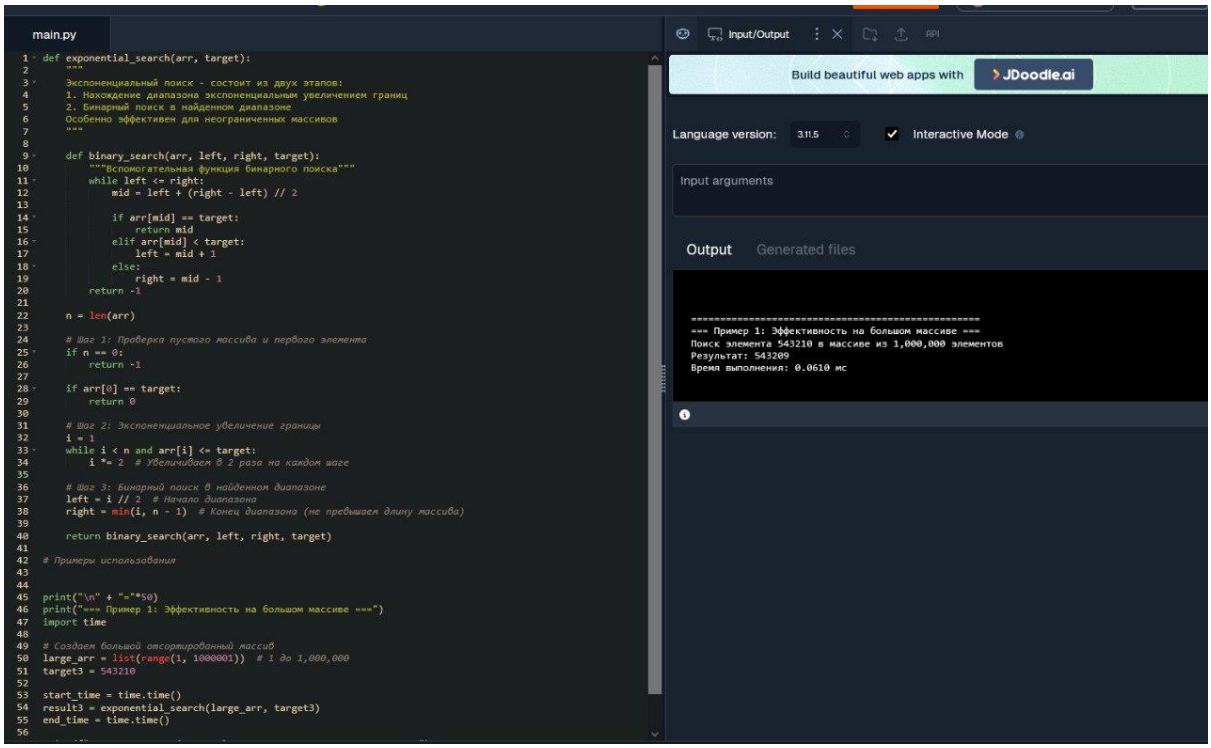



```python
def ternary_search(arr, target):
    """
    Тернарный поиск - делит массив на три части
    Рекурсивная реализация
    """
    def ternary_recursive(left, right):
        if left > right:
            return -1

        # Шаг 1: Делим массив на три части
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3

        # Шаг 2: Проверяем точки деления
        if arr[mid1] == target:
            return mid1
        if arr[mid2] == target:
            return mid2

        # Шаг 3: Определяем, в какой части продолжать поиск
        if target < arr[mid1]:
            return ternary_recursive(left, mid1 - 1)
        elif target > arr[mid2]:
            return ternary_recursive(mid2 + 1, right)
        else:
            return ternary_recursive(mid1 + 1, mid2 - 1)

    return ternary_recursive(0, len(arr) - 1)

def ternary_search_iterative(arr, target):
    """
    Тернарный поиск - итеративная реализация
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        # Шаг 1: Делим массив на три части
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3

        # Шаг 2: Проверяем точки деления
        if arr[mid1] == target:
            return mid1
        if arr[mid2] == target:
            return mid2

        # Шаг 3: Определяем, в какой части продолжать поиск
        if target < arr[mid1]:
            right = mid1 - 1
        elif target > arr[mid2]:
            left = mid2 + 1
        else:
```

Exponential_search


Online Python 3 IDE
Execute
Assembler (T...
Login
main.py
Input/Output
Build beautiful web apps with
JDoodle.ai
Language version: 3.11.5
Interactive Mode
Input arguments
Output
Generated files
Сравнение алгоритмов поиска:
Размер массива: 1,000,000
Целевой элемент: 54321
Jump Search:
Позиция: 54320
Время: 0.2499 мс
Exponential Search:
Позиция: 54320
Время: 0.0246 мс
Ternary Search:
Позиция: 54320
Время: 0.0191 мс

Сравнение


main.py
Input/Output
Build beautiful web apps with
JDoodle.ai
Language version: 3.11.5
Interactive Mode
Input arguments
Output
Generated files
==
=== Пример 1: Эффективность на большом массиве ===
Поиск элемента 543210 в массиве из 1,000,000 элементов
Результат: 543209
Время выполнения: 0.0610 мс