Лабораторная работа №9-10

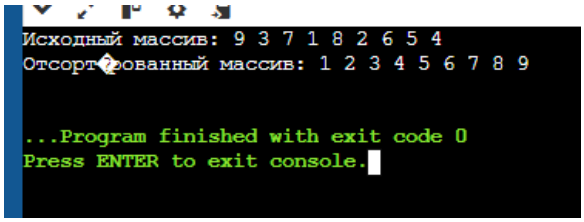
«Алгоритмы. Работа в онлайн-IDE jdoodle.com с алгоритмами сортировки и поиска»

Алгоритмы сортировки

1. Блочная (корзинная) сортировка

Блочная сортировка (bucket sort) — это алгоритм, который распределяет элементы исходного массива на несколько блоков (корзин) в зависимости от диапазона их значений. После этого каждая корзина сортируется отдельно (либо тем же методом рекурсивно, либо другим алгоритмом, например, вставками), и отсортированные корзины объединяются обратно в один массив.
Пример (C++):

```
#include <iostream>
#include <vector>
#include <algorithm> // для std::sort

// Функция для сортировки массива с помощью блочной сортировки
void blockSort(std::vector<int>& arr, int blockSize) {
    int n = arr.size();

    // Сортируем каждый блок отдельно
    for (int i = 0; i < n; i += blockSize) {
        int end = std::min(i + blockSize, n);
        std::sort(arr.begin() + i, arr.begin() + end);
    }

    // Объединяем отсортированные блоки
    // В данном простом примере, после сортировки отдельных блоков,
    // можно дополнительно выполнить полное слияние или другую стратегию.
    // Для простоты — просто повторная сортировка всего массива:
    std::sort(arr.begin(), arr.end());
}

int main() {
    std::vector<int> arr = {9, 3, 7, 1, 8, 2, 6, 5, 4};
    int blockSize = 3;

    std::cout << "Исходный массив: ";
    for (int num : arr) std::cout << num << " ";
    std::cout << std::endl;

    blockSort(arr, blockSize);

    std::cout << "Отсортированный массив: ";
    for (int num : arr) std::cout << num << " ";
    std::cout << std::endl;

    return 0;
}
```



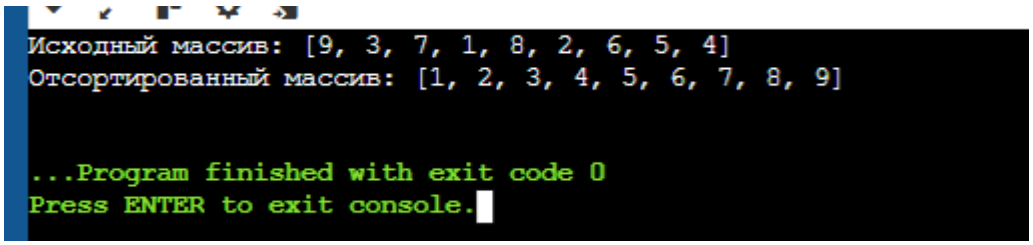
Общая временная сложность составляет O(n * log n), а пространственная сложность — O(1).

2. Блинная сортировка

Блинная сортировка (pancake sort) — это метод, основанный на операции переворота (reversal) части массива. В отличие от большинства алгоритмов, которые обменивают соседние элементы или выбирают опорные точки, здесь единственная разрешённая операция — переворот префикса массива до выбранного индекса.

Пример (Python):

```
def pancake_sort(arr):
    n = len(arr)
    for curr_size in range(n, 1, -1):
        # Находим максимум в массиве
        max_idx = arr.index(max(arr[:curr_size]))
        # Переворачиваем массив до максимум включая его
        if max_idx != 0:
            arr[:max_idx+1] = arr[:max_idx+1][::-1]
        # Переворачиваем весь оставшийся массив
        arr[:curr_size] = arr[:curr_size][::-1]

# Пример использования
array = [9, 3, 7, 1, 8, 2, 6, 5, 4]
print("Исходный массив:", array)
pancake_sort(array)
print("Отсортированный массив:", array)
```



Временная сложность: O(n^2)

Пространственная сложность: O(1).

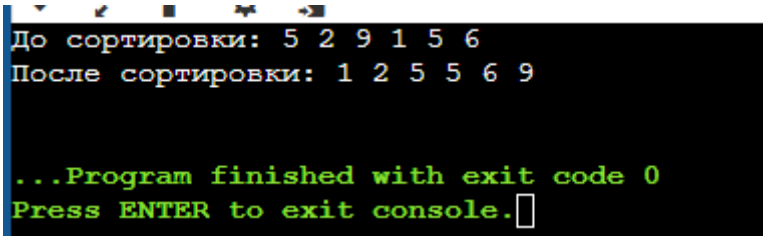
3. Сортировка бусинами (гравитационная)

Сортировка бусинами (bead sort), также известная как гравитационная сортировка, моделирует естественное поведение бусин, падающих вниз под действием гравитации.

Пример (C++):

```
#include <iostream>
#include <vector>

void bubbleSort(std::vector<int>& beads) {
  bool swapped;
  for (size_t i = 0; i < beads.size() - 1; ++i) {
    swapped = false;
    for (size_t j = 0; j < beads.size() - i - 1; ++j) {
      if (beads[j] > beads[j + 1]) {
        std::swap(beads[j], beads[j + 1]);
        swapped = true;
      }
    }
    if (!swapped) break;
  }
}

int main() {
  std::vector<int> beads = {5, 2, 9, 1, 5, 6};

  std::cout << "До сортировки: ";
  for (int bead : beads) {
    std::cout << bead << " ";
  }
  std::cout << std::endl;
  bubbleSort(beads);

  std::cout << "После сортировки: ";
```

```
    for (int bead : beads) {
        std::cout << bead << " ";
    }
    std::cout << std::endl;

    return 0;
}
```



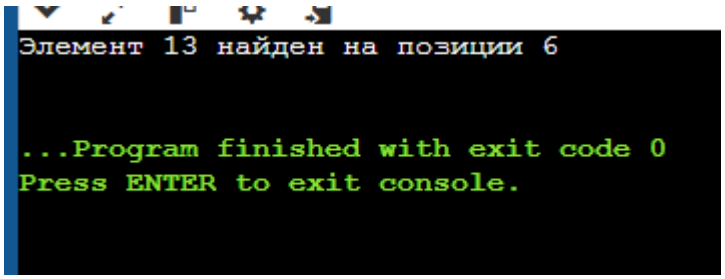
Временная сложность: $O(n^2)$ в худшем случае или $O(n)$
Пространственная сложность: $O(1)$

4. Поиск скачками (Jump Search)

Поиск скачками (Jump Search) - это оптимизация линейного поиска для отсортированных массивов, при которой алгоритм «прыгает» через определённое число элементов (обычно через n), чтобы быстро сузить диапазон поиска, а затем выполняет линейный поиск внутри найденного диапазона.

Пример (Python):

```
import math

def jump_search(arr, x):
    n = len(arr)
    step = int(math.sqrt(n))
    prev = 0

    # Сделать прыжки вперед, пока не найдем блок, где x может находиться
    while prev < n and arr[min(step, n) - 1] < x:
        prev = step
        step += int(math.sqrt(n))
        if prev >= n:
            return -1  # Не найден

    # Линейный поиск внутри найденного блока
    while prev < min(step, n):
        if arr[prev] == x:
            return prev  # Индекс найден
        prev += 1

    return -1  # Не найдено

# Пример использования:
arr = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
x = 13

index = jump_search(arr, x)
if index != -1:
    print(f"Элемент {x} найден на позиции {index}")
else:
    print(f"Элемент {x} не найден в массиве")
```



Временная сложность: O(√n) + O(√n) = O(√n)

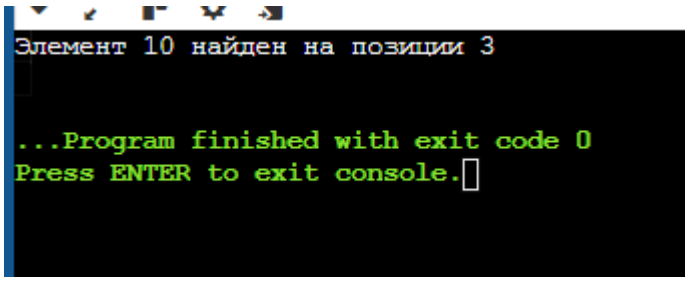
Пространственная сложность: O(1)

5. *Экспоненциальный поиск (Exponential Search)*

Экспоненциальный поиск сочетает идеи последовательного и бинарного поиска, позволяя быстро определить диапазон, в котором может находиться искомое значение, а затем применяет двоичный поиск в этом диапазоне.

Пример (C++):

```
#include <iostream>
#include <vector>
#include <algorithm> // для std::binary_search и std::lower_bound

int exponentialSearch(const std::vector<int>& arr, int x) {
    if (arr.empty()) return -1;

    // Если элемент в первом элементе
    if (arr[0] == x) return 0;

    // Найти диапазон, где может находиться x
    int i = 1;
    int n = arr.size();

    while (i < n && arr[i] <= x) {
        i *= 2;
    }

    // Выполнить бинарный поиск в определенном диапазоне
    int left = i / 2;
    int right = std::min(i, n - 1);

    auto it = std::lower_bound(arr.begin() + left, arr.begin() + right + 1, x);
    if (it != arr.end() && *it == x) {
        return std::distance(arr.begin(), it);
    } else {
        return -1; // Не найден
    }
}

int main() {
    std::vector<int> arr = {2, 3, 4, 10, 40, 50, 60, 70, 80, 90};
    int x = 10;

    int index = exponentialSearch(arr, x);
    if (index != -1)
        std::cout << "Элемент " << x << " найден на позиции " << index << std::endl;
    else
        std::cout << "Элемент " << x << " не найден" << std::endl;

    return 0;
}
```



Временная сложность: O(log n)

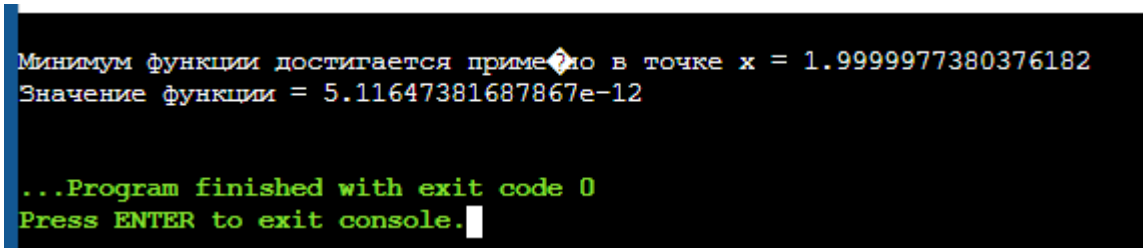
Пространственная сложность: O(1)

6. *Тернарный поиск*

Тернарный поиск — это метод деления диапазона поиска на три части (в отличие от двух в бинарном) и рекурсивного сужения диапазона до тех пор, пока не будет найден элемент или экстремум функции (в зависимости от задачи).

Пример (Python):

```
def ternary_search(f, left, right, eps=1e-5):
    while right - left > eps:
        m1 = left + (right - left) / 3
        m2 = right - (right - left) / 3
        f1 = f(m1)
        f2 = f(m2)

        if f1 > f2:
            left = m1
        else:
            right = m2
    return (left + right) / 2

# Пример функции: parabola с минимумом в 2
def func(x):
    return (x - 2) ** 2

min_x = ternary_search(func, 0, 4)
print(f"Минимум функции достигается примерно в точке x = {min_x}")
print(f"Значение функции = {func(min_x)}")
```



Пространственная сложность: O(log n)

Временная сложность: O(1)