МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра информатики и программирования

АНАЛИЗ СЛОЖНОСТИ БЫСТРОЙ И ПИРАМИДАЛЬНОЙ СОРТИРОВОК

ОТЧЕТ

студентки 2 курса 211 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета компьютерных наук и информационных технологий
Никитенко Яны Валерьевны

Саратов 2025

СОДЕРЖАНИЕ

1 Быстрая сортировка

1.1 Текст программы

```
  //  Для опорного элемента и разделеения массива элемента
int partition(vector<int>& arr, int low, int high) {


    int pivot = arr[high]; // Выбор опорной точки


    int i = low - 1; // Индекс наименшьего элемента распологается
    справа от опорного элемнта

        // Прохождение arr[low..high] и перемещщние все меньших
        // элементы слева. Элементы от low до  high
        // i уменьшается после каждой итерации
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i + 1], arr[high]); // Передвижение опорного элемента
    после меньших элементов
    return i + 1;   // Возвращение позиций
}
//

// Функция быстрой сортировки
void quickSort(vector<int>& arr, int low, int high) {
    while (low < high) {
      int pi = partition(arr, low, high); // Возвращает индекс опорного элемента

        // Рекурсивные вызовы для меньших и больших элементов
```

```
        if (pi - low < high - pi) {
            quickSort(arr, low, pi - 1);
            low = pi + 1; // Избегаем переполнения стека
        }
        else {
            quickSort(arr, pi + 1, high);
            high = pi - 1; // Избегаем переполнения стека
        }
    }
}
//
```

1.2 Анализ сложности

— **Лучший случай** (сбалансированное разбиение):

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

По основной теореме: $T(n) = O(n \log n)$

— **Худший случай** (несбалансированное разбиение):

$$T(n) = T(n-1) + O(n)$$

Решение: $T(n) = O(n^2)$

— **Средний случай**: $T(n) = O(n \log n)$

Объяснение:

— Функция $\text{partition}()$ имеет сложность $O(n)$, так как проходит по всем элементам подмассива один раз

— В лучшем случае массив делится пополам, создавая $\log n$ уровней рекурсии

— На каждом уровне выполняется работа $O(n)$, итого $O(n \log n)$

— В худшем случае глубина рекурсии n, на каждом уровне работа $O(n)$, итого $O(n^2)$

— В представленной реализации используется оптимизация хвостовой рекурсии, которая уменьшает глубину стека, но не меняет асимптотическую сложность

2 Пирамидальная сортировка

2.1 Текст программы

```
// Для создания кучи поддерева с корнем в узле i, который
// индекс в arr[]. n — размер кучи
void heapify(int arr[], int n, int i) {
    int largest = i; // Инициализируем наибольший элемент как корень
    int l = 2 * i + 1; // левый = 2*i + 1
    int r = 2 * i + 2; // правый = 2*i + 2

    // Если левый дочерний элемент больше корня
    if (l < n && arr[l] > arr[largest])
        largest = l;
    //

    // Если правый дочерний элемент больше,
    чем наибольший элемент на данный момент
    if (r < n && arr[r] > arr[largest])
        largest = r;
    //

    // Если наибольший элемент не корень
    if (largest != i) {
        swap(arr[i], arr[largest]); // Перестановка
        heapify(arr, n, largest); // Рекурсивная группировка
        соответствующего поддерево
    }
    //
}
//

// Основная функция сортировки кучи
void heapSort(int arr[], int n) {
    // Построение кучи (перегруппировка массива)
    for (int i = n / 2 - 1; i >= 0; i--)
```

```
        heapify(arr, n, i);

    // Извлечение элементов из кучи
    for (int i = n - 1; i >= 0; i--) {
        swap(arr[0], arr[i]); // Перемещение текущего элемента

        // Вызов на усеньшенной куче
        heapify(arr, i, 0);
    }
}
//
```

2.2 Анализ сложности

Анализ функций:

- **Функция** heapify()**:**
  - В худшем случае элемент "просеивается"от корня до листа
  - Высота бинарной кучи: $O(\log n)$
  - Сложность heapify(): $O(\log n)$
- **Построение кучи:**
  - Выполняется для элементов от $n/2 - 1$ до 0
  - Суммарная сложность: $O(n)$
- **Процесс сортировки:**
  - Выполняется n извлечений максимального элемента
  - Каждое извлечение требует heapify() за $O(\log n)$
  - Сложность: $n \times O(\log n) = O(n \log n)$

Общая сложность:

$$T(n) = O(n) + O(n \log n) = O(n \log n)$$

Особенности:

- Сложность всегда $O(n \log n)$ независимо от входных данных
- Не требует дополнительной памяти (in-place)

— Неустойчивая сортировка