МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

АНАЛИЗ СЛОЖНОСТИ БЫСТРОЙ И ПИРАМИДАЛЬНОЙ СОРТИРОВОК

ОТЧЁТ

студента 2 курса 251 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Толстова Роберта Сергеевича

Проверено:

доцент, к. ф.-м. н. _____ М. И. Сафрончик

Саратов 2025

СОДЕРЖАНИЕ

1 Быстрая сортировка

```
#include <algorithm>

int partition(int* arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quicksort(int* arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}
```

В лучшем и среднем случаях: при удачном выборе опорного элемента (обычно середина массива) алгоритм делит массив на две примерно равные части. В этом случае временная сложность:

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

В худшем случае: при неудачном выборе опорного элемента (например, минимум или максимум):

$$T(n) = T(n-1) + O(n) \Rightarrow T(n) = O(n^2)$$

2 Пирамидальная сортировка

```
#include <algorithm>

void heapify(int* arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapsort(int* arr, int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        std::swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

В худшем случае рекурсивный вызов идет по высоте heap – O(log n).

Рассмотрим временную сложность построения кучи. Мы вызываем `heapify` почти для n/2 элементов. Общей же сложностью будет O(n). Это установлено по свойству кучи (сумма всех узлов)

После построения будем извлекать элементы из кучи для построения отсортированного массива. Второй цикл выполняется n-1 раз, каждый раз вызывается `heapify` O(log n).

Общая временная сложность: $O(n \cdot \log n)$.