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

Общая сложность: Для 2-х вложенных циклов общая сложность равна: (t + m) = O(n), Для внешнего while сложность: O(kn), где n – общее кол-во элементов, k – кол-во повторений внешнего цикла. Тогда общая сложность по правилу суммы: O(max(kn,1,1)) = O(kn). Если массив отсортирован, то в лучшем случае мы просто пройдёмся по его n элементам. Сложность равна O(nlogn), так как мы делим каждый раз при рекурсивном вызове массив пополам, и всего таких случаев log2 n. Согласно первой теореме:

$$t(n) = \{c,\ \text{если}\ n = 1\ \alpha t\left(\frac{n}{k}\right) + bn^{\tau},\ \text{если}\ n > 1$$

построим реккурентное соотношение:

$$t(n) = \begin{cases} c \\ 2T(\frac{n}{2}) + bn \end{cases}$$

Где a = 2 – кол-во подзадачи, порождаемых рекурсивной веткой, n/k – размер подзадач, k = 2 – постоянная величина.

Трудоемкость рекурсивного перехода имеет порядок $O(n), \tau = 1$. Значит по следствию теоремы для лучшего случая $t(n) = O(n^{\tau} log_k n) = O(nlogn)$.

В худшем случае опорный элемент каждый раз оказывается самым большим или самым маленьким элементом, что приводит к неравномерному разделению массива.

Каждый раз опорный элемент оказывается таким, что один из подмассивов пуст, а другой содержит все элементы, кроме опорного. Если массив изначально отсортирован (или отсортирован в обратном порядке), каждый раз после разделения один из подмассивов будет содержать n - 1 элементов.

Таким образом, количество уровней рекурсии будет n. На каждом уровне рекурсии мы выполняем O(n) операций по разделению массива и сравнению элементов. В сумме на всех уровнях это дает O(n_2).

Для худшего случая получим t(n) = O(n_2).

Быстрая сортировка эффективна, когда по обеим сторонам от опорного элемента лежит равное количество элементов.

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

Сортировка основана на построении кучи. Свойства кучи: Узел всегда больше своих потомков; На последнем уровне элементы располагаются слева направо, пока онине закончатся.

Т.к. куча – это бинарное дерево, то его высота не больше, чем $log_2 N$, а значит временная сложность функции построения кучи равна O(logN).

В самом цикле сортировки мы проходимся по N элементов, значит общая сложность алгоритма O(N logN). Эффективность алгоритма уменьшается, если все большие значения находятся в одной части пирамиды.