Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1

Выполнил студент группы .....КС-38.....(Нергарян Геворг Гарегинович)
Ссылка на репозиторий: .....(https://github.com/MUCTR-IKT-CPP/Nergaryan_KC-38_Algos)

Приняли: .....Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович
.....Лобанов Алексей Владимирович
.....Крашенинников Роман Сергеевич

Дата сдачи: ……10.03.2023……..

Оглавление

Описание задачи.

Вариант 1.

Необходимо реализовать метод быстрой сортировки.

Для реализованного метода сортировки необходимо провести серию тестов для всех значений N из списка (1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000), при этом:

- в каждом тесте необходимо по 20 раз генерировать вектор, состоящий из N элементов
- каждый элемент массива заполняется случайным числом с плавающей запятой от -1 до 1

На основании статьи реализовать проверки негативных случаев и устроить на них серии тестов аналогичные второму пункту:

- Отсортированный массив
- Массив с одинаковыми элементами
- Массив с максимальным количеством сравнений при выборе среднего элемента в качестве опорного
- Массив с максимальным количеством сравнений при детерминированном выборе опорного элемента

При работе сортировки подсчитать количество вызовов рекурсивной функции, и высоту рекурсивного стека. Построить график худшего, лучшего, и среднего случая для каждой серии тестов.

Для каждой серии тестов построить график худшего случая.

Подобрать такую константу c, что бы график функции c * n * log(n) находился близко к графику худшего случая, если возможно построить такой график.

Проанализировать полученные графики и определить есть ли на них следы деградации метода относительно своей средней сложности.

Описание метода/модели.

Метод быстрой сортировки (также известный как quicksort) - это алгоритм сортировки, который использует стратегию "разделяй и властвуй".

Он работает следующим образом:

1. Выбирается опорный элемент из массива (обычно это первый, последний или средний элемент).
2. Остальные элементы массива сравниваются с опорным элементом и разделяются на две группы: элементы, меньшие опорного, и элементы, большие опорного.
3. Рекурсивно повторяется для каждой из двух групп, пока каждый элемент не станет отдельным подмассивом.
4. Результаты объединяются, чтобы получить отсортированный массив.

Быстрая сортировка - один из самых эффективных алгоритмов сортировки. Он имеет среднее время выполнения O(n log n) и часто используется в программных библиотеках и приложениях. Однако в худшем случае время выполнения может быть O(n^2), если выбранный опорный элемент оказывается наименьшим или наибольшим элементом в массиве, что приводит к неэффективной работе алгоритма.

Выполнение задачи.

Использовался язык C++. Время измеряли в микросекундах.

Код:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <cassert>
#include <cmath>

using namespace std;
vector<double> quicksort(vector<double>& arr, int& kolvo_recurs, int& glubina_steka, int cur_depth = 0) {  //cur_depth используется для отслеживания текущей глубины рекурсии.
    kolvo_recurs++;
    if (arr.size() <= 1) {    //если размер массива не больше 1, то он уже отсортирован и возвращается без изменений
        return arr;
    }
    double pivot = arr[arr.size() / 2];  //опорный элемент (pivot) из середины массива
    vector<double> left, middle, right;  //три вектора для элементов, которые меньше, равны и больше опорного элемента
    for (double x : arr) {
        if (x < pivot) {
            left.push_back(x);
        }
        else if (x == pivot) {
            middle.push_back(x);
        }
        else {
            right.push_back(x);
        }
    }
    int cur_depth_left = cur_depth + 1;
    vector<double> sorted_left = quicksort(left, kolvo_recurs, glubina_steka, cur_depth_left);
    glubina_steka = max(glubina_steka, cur_depth_left); //рекурсивно вызывает саму себя для двух подмассивов, созданных из элементов, которые меньше и больше опорного элемента
    int cur_depth_right = cur_depth + 1;
    vector<double> sorted_right = quicksort(right, kolvo_recurs, glubina_steka, cur_depth_right);
    glubina_steka = max(glubina_steka, cur_depth_right);
    vector<double> result;
    result.reserve(sorted_left.size() + middle.size() + sorted_right.size());
    result.insert(result.end(), sorted_left.begin(), sorted_left.end());
    result.insert(result.end(), middle.begin(), middle.end());
    result.insert(result.end(), sorted_right.begin(), sorted_right.end());
    return result; // функция объединяет три отсортированных вектора (левый, средний и правый) в один и возвращает его
}


int main() {
    setlocale(LC_ALL, "Russian");
    int min_r, avg_r, max_r; // переменные для подсчета количества вызовов
    vector<int> N_values = { 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000 };
    for (int N : N_values) {
        int glubina_steka = 0, kolvo_recurs = 0, time = 0;
        min_r = 9999999, avg_r = 0, max_r = 0;
        for (int i = 0; i < 20; i++) {
```

```
        auto start = std::chrono::high_resolution_clock::now();//начало времени
        glubina_steka = 0, kolvo_recurs = 0;
        vector<double> arr(N);
        generate(arr.begin(), arr.end(), []() { return ((double)rand() / RAND_MAX) * 2 - 1; }); //используется лямбда-функция без аргументов, которая генерирует случайное число в диапазоне от -1 до 1
        vector<double> sorted_arr = quicksort(arr, kolvo_recurs, glubina_steka);  //отсортированные по возрастанию
        bool is_sorted = is_sorted_until(sorted_arr.begin(), sorted_arr.end()) == sorted_arr.end(); //Проверяется, отсортирован ли "sorted_arr" по возрастанию
        if (!is_sorted) {
            cout << "Сортировка для N=" << N << " и итерации " << i << " была неверной!" << endl;
        }
        // Отсортированный массив
        vector<double> sorted_arr_copy = sorted_arr;// создается копия отсортированного массива "sorted_arr"
        sort(sorted_arr_copy.begin(), sorted_arr_copy.end()); //снова сортировка
        bool is_sorted_copy = is_sorted_until(sorted_arr_copy.begin(), sorted_arr_copy.end()) == sorted_arr_copy.end();
        if (!is_sorted_copy) { //проверка
            cout << "Негативный тест: Сортировка отсортированного массива для N=" << N << " и итерации " << i << " была неверной!" << endl;
        }
        // Массив с одинаковыми элементами
        vector<double> same_arr(N, 1.0);
        vector<double> same_arr_sorted = quicksort(same_arr, kolvo_recurs, glubina_steka);
        bool is_same_sorted = is_sorted_until(same_arr_sorted.begin(), same_arr_sorted.end()) == same_arr_sorted.end();
        if (!is_same_sorted) {
            cout << "Негативный тест: Сортировка массива с одинаковыми элементами для N=" << N << " и итерации " << i << " была неверной!" << endl;
        }
        // Массив с максимальным количеством сравнений при выборе среднего элемента в качестве опорного
        vector<double> max_comparison_arr(N);
        generate(max_comparison_arr.begin(), max_comparison_arr.end(), []() { return 1.0; }); //лямбда-функция без аргументов, которая всегда возвращает 1.0
        vector<double> max_comparison_sorted_arr = quicksort(max_comparison_arr, kolvo_recurs, glubina_steka);  //сортировка по возрастанию
        bool is_max_comparison_sorted = is_sorted_until(max_comparison_sorted_arr.begin(), max_comparison_sorted_arr.end()) == max_comparison_sorted_arr.end(); //Проверяется, отсортирован ли вектор по возрастанию
        if (!is_max_comparison_sorted) {
            cout << "Негативный тест: Сортировка массива с максимальным количеством сравнений для N=" << N << " и итерации " << i << " была неверной!" << endl;
        }
        //Массив с максимальным количеством сравнений при детерминированном выборе опорного элемента
        vector<double> dete_arr(N);
        generate(dete_arr.begin(), dete_arr.end(), []() { return ((double)rand() / RAND_MAX) * 2 - 1; });
        vector<double> dete_sorted_arr = quicksort(dete_arr, kolvo_recurs, glubina_steka);
        bool dete_is_sorted = is_sorted_until(dete_sorted_arr.begin(), dete_sorted_arr.end()) == dete_sorted_arr.end();
        if (!dete_is_sorted) {
            cout << "Негативный тест: Сортировка массива с детерминированным выбором опорного элемента для N=" << N << " была неверной!" << endl;
        }
        if (min_r > kolvo_recurs) { min_r = kolvo_recurs; }
        if (max_r < kolvo_recurs) { max_r = kolvo_recurs; }
        avg_r += kolvo_recurs;
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        if (time < duration.count()) { time = duration.count(); }
    }
        cout << "Время худшего выполнения: " << time << " микросекунд" << endl;
    cout << "Лучший случай:" << min_r << "      Средний случай:" << avg_r/20 << "      Худший случай:" << max_r << endl;
    cout << "Глубина стека для N=" << N << ":" << glubina_steka << "    Количество рекурсий для N=" << N << ":" << avg_r << endl;
    cout << "Тесты для N=" << N << " успешно пройдены!" << endl << endl << endl;
  }
  return 0;
}
```

Реализован метод быстрой сортировки для всех значений N. В каждом тесте генерируется 20 векторов, заполненных случайным числом с плавающей запятой от -1 до 1. Провел негативные тесты по тому же алгоритму.

Вывод:

Программа выводит худшее время теста для каждого N, минимальное, среднее и максимальное количество вызова рекурсии для теста, для всего N и высоту рекурсивного стека.

```
Время худшего выполнения: 14520 микросекунд
Лучший случай:2616      Средний случай:2643      Худший случай:2672
Глубина стека для N=1000:21    Количество рекурсий для N=1000:52874
Тесты для N=1000 успешно пройдены!


Время худшего выполнения: 29000 микросекунд
Лучший случай:5182      Средний случай:5225      Худший случай:5264
Глубина стека для N=2000:26    Количество рекурсий для N=2000:104516
Тесты для N=2000 успешно пройдены!


Время худшего выполнения: 58078 микросекунд
Лучший случай:10116      Средний случай:10203      Худший случай:10274
Глубина стека для N=4000:27    Количество рекурсий для N=4000:204064
Тесты для N=4000 успешно пройдены!


Время худшего выполнения: 114004 микросекунд
Лучший случай:19430      Средний случай:19538      Худший случай:19634
Глубина стека для N=8000:31    Количество рекурсий для N=8000:390762
Тесты для N=8000 успешно пройдены!


Время худшего выполнения: 218106 микросекунд
Лучший случай:35710      Средний случай:35901      Худший случай:36158
Глубина стека для N=16000:33    Количество рекурсий для N=16000:718036
Тесты для N=16000 успешно пройдены!


Время худшего выполнения: 405988 микросекунд
Лучший случай:61392      Средний случай:61512      Худший случай:61760
Глубина стека для N=32000:34    Количество рекурсий для N=32000:1230248
Тесты для N=32000 успешно пройдены!


Время худшего выполнения: 709828 микросекунд
Лучший случай:93880      Средний случай:94152      Худший случай:94638
Глубина стека для N=64000:38    Количество рекурсий для N=64000:1883040
Тесты для N=64000 успешно пройдены!


Время худшего выполнения: 1141290 микросекунд
Лучший случай:121620      Средний случай:121777      Худший случай:121940
Глубина стека для N=128000:39    Количество рекурсий для N=128000:2435558
Тесты для N=128000 успешно пройдены!
```

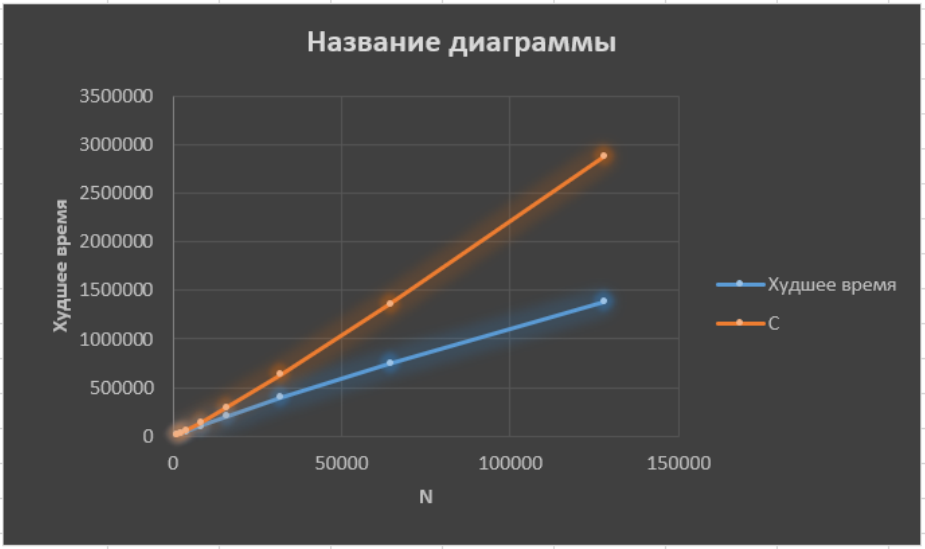
Графики:

График худшего, лучшего, и среднего случая для каждой серии тестов. Из-за незначительности отличий на графике хорошо видно только худший случай, но остальные за ней.

N	Лучший	Средний	Худший
1000	2332	2348	2372
2000	4632	4660	4686
4000	9164	9201	9230
8000	17882	17965	18034
16000	34242	34348	34462
32000	63348	63563	63744
64000	112290	112566	112796
128000	191772	191928	192154

График худшего случая по времени для каждого теста. График функции c * n * log(n) и сама константа.

N	Худшее время	C
1000	13403	13200
2000	26321	29049,1
4000	52736	63396,3
8000	107010	137389
16000	211440	295970
32000	400911	634325
64000	749551	1353420
128000	1380597	2876381
C=	4,4	



Заключение.

Если данные, которые нужно отсортировать, уже упорядочены или имеют определенный порядок, то быстрая сортировка может замедлиться и работать со сложностью O(n^2), что значительно хуже, чем средняя сложность O(n log n). Это происходит потому, что в таких случаях первоначально выбранный опорный элемент может оказаться крайним элементом в массиве, и тогда каждая итерация сортировки будет уменьшать размер обрабатываемого массива всего на один элемент. Однако, если опорный элемент выбирается случайным образом, вероятность того, что массив уже

отсортирован или почти отсортирован, очень мала, и быстрая сортировка будет работать со средней сложностью O(n log n).