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 № 2

Вариант 3

Выполнила студентка группы .....КС-33..... Георгиевская Анастасия Игоревна
Ссылка на репозиторий: .....
https://github.com/MUCTR-IKT-CPP/GeorgievskayaAA_33_alg/tree/main/lab%202

Приняли: .....Пысин Максим Дмитриевич
.....Лобанов Алексей Владимирович

Дата сдачи: 05.03.2025

Оглавление

Описание задачи.

Задание:

Используя предыдущий код посерийного выполнения алгоритма сортировки и измерения времени требуется реализовать метод пирамидальной сортировки.

- Реализовать проведения тестирования алгоритма сериями расчетов для измерения параметров времени.

  За один расчет выполняется следующие операции:

  i. Генерируется массив случайных значений
  ii. Запоминается время начала расчета алгоритма сортировки
  iii. Выполняется алгоритм сортировки
  iv. Вычисляется время затраченное на сортировку: текущее время - время начала
  v. Сохраняется время для одной попытки. После этого расчет повторяется до окончания серии.
     - Алгоритм вычисляется 8 сериями по 20 раз за серию.
     - Алгоритм в каждой серии вычисляется для массива размером M. (1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000)
     - Массив заполняется значения числами с плавающей запятой в интервале от -1 до 1.
     - Для серии запоминаются все времена которые были замерены

- При работе сортировки подсчитать:
  i. общее количество вызовов функции построения кучи
  ii. количество вызовов внутренней функции построения кучи(вызов внутри функции формирования кучи)
  iii. времени исполнения сортировки
- По полученным данным времени построить графики зависимости времени от числа элементов в массиве:
  i. Совмещенный график наихудшего времени выполнения сортировки и сложности алгоритма указанной в нотации О большое.

ii. Совмещенный график среднего, наилучшего и наихудшего времени исполнения.
iii. Совмещенный график среднего, наилучшего и наихудшего глубины рекурсии.
iv. Совмещенный график среднего по серии количества вызовов функции построения кучи и количества вызовов внутренней функции.
v. График среднего процентного соотношения вызовов внутренней функции к общему вызову функции.

- По результатом расчетов оформляется отчет по предоставленной форме, в отчете:
  i. Приводится описание алгоритма.
  ii. Приводится описания выполнения задачи (Описание кода и специфических элементов реализации)
  iii. Приводятся выводы (Графики и их анализ). Требуется ответить на вопрос о поведении алгоритма изученного в процессе выполнения лабораторной работы и зафиксировать его особенности.

Описание пирамидальной сортировки.

Сортировка кучей — это алгоритм сортировки, основанный на структуре данных «куча». Алгоритм использует бинарную кучу (обычно максимальную или минимальную), чтобы эффективно сортировать массив.

Этапы работы:

1. Построение кучи: выполняется операция «просеивания» элементов с конца массива до корня
2. Извлечение максимального элемента: После построения кучи, максимальный или минимальный элемент будет находиться в корне. Этот элемент обменивается с последним элементом в массиве. Затем нужно уменьшить размер кучи и повторить процесс «просеивания» вниз, чтобы восстановить свойства кучи.
3. Повторение: Повторяем процесс извлечения максимального (или минимального) элемента и восстановления кучи до тех пор, пока все элементы не окажутся отсортированы.

Оценка сложности алгоритма

Время выполнения:

1. Лучший случай — O(n log n). Даже если элементы уже отсортированы, построение кучи и дальнейшие операции все равно требуют логарифмических затрат на каждом шаге.
2. Средний и худший случаи — O(n log n). Эти случаи также требуют полной перестройки кучи и извлечения элементов, что по времени аналогично лучшему.

Преимущества алгоритма

1. Гарантированное время работы O(n log n) — независимо от начальной упорядоченности массива, алгоритм всегда работает за время O(n log n).

Недостатки алгоритма

1. Неустойчивость — алгоритм сортировки кучей является неустойчивым, что означает, что равные элементы могут изменять свой порядок относительно друг друга.

Выполнение задачи.

Для выполнения задачи лабораторной был использован язык C.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

- Подключаемые библиотеки
    - <stdio.h> — стандартная библиотека для ввода-вывода, включает функции для работы с файлами (fopen, fclose, fprintf), а также функции вывода в консоль (printf).
    - <stdlib.h> — библиотека для работы с динамической памятью, генерацией случайных чисел и другими утилитами (например, malloc, rand, free).
    - <time.h> — библиотека для работы с временем. Здесь она используется для отслеживания времени выполнения программы с помощью функции clock().

```
void generation(double arr[], int n) {
   int i = 0;
    for(i = 0; i < n; i++) {
        arr[i] = (2.0 * rand() / RAND_MAX) - 1.0;
    }
}
```

- Функция generation
    - Вход: пустой массив заданной длины, количество элементов
    - Назначение: Генерация случайных чисел в диапазоне от -1 до 1 для массива заданной длины.
    - Как работает: Функция использует стандартную функцию rand(), чтобы получить случайное число в диапазоне от 0 до RAND_MAX, а затем с помощью математической операции масштабирует его в диапазон от -1 до 1.
    - Результат: заполненный числами от -1 до 1 массив

```
void shift(double* a, double* b){
   double temp = *a;
   *a = *b;
   *b = temp;
}
```

- Функция shift
    - Вход: указатель на переменные.
    - Назначение: Обмен элементов.
    - Как работает: Обмен осуществляется при помощи дополнительной переменной, поочередно присваивая значения.
    - Результат: поменянные местами элементы.

```
void heapify(double arr[], int m, int i,
                  unsigned long long int* shift_count,
                  unsigned long long int* recursive) {
    int largest = i, left = 2 * i + 1, right = 2 * i + 2;
    (*shift_count)++;

    if (left < m && arr[left] > arr[largest])
        largest = left;
    if (right < m && arr[right] > arr[largest])
         largest = right;
    if( largest != i){
         shift(&arr[i], &arr[largest]);
         (*recursive)++;
         heapify(arr, m, largest, shift_count, recursive);
    }
}
```

- Функция heapify:
    - Вход: массив чисел, количество элементов, индекс текущего узла, количество обших и рекурсивных вызовов функции.
    - Назначение: Образование кучи в порядке возрастания.
    - Как работает: поддерживает, что каждый родительский элемент больше дочернего
    - Результат: отсортированный массив.

```
void heap(double arr[], int m, unsigned long long int* shift_count, unsigned
long long int* recursive){
    *recursive = 0;
    *shift_count = 0;
    int i, j = 0;

    for (i = m / 2 - 1; i >= 0; i--){
         heapify(arr, m, i, shift_count, recursive);
         (*shift_count)++;
    }

    for(j = m - 1; i >= 1; i--){
         shift(&arr[0], &arr[i]);

         heapify(arr, i, 0, shift_count, recursive);
         (*shift_count)++;
    }
}
```

- Функция heap
    - Вход: массив элементов, кол-во элементов, указатели на счетчики.
    - Назначение: Сортировка кучей
    - Как работает: Строится максимальная куча. На каждом шаге алгоритма размер массива уменьшается, извлекаемый элемент ставится на последнее место.
    - Результат Отсортированный массив

```
  void measure(int m, int s) {
      FILE *f = fopen("times.txt", "a");
      if (!f) {
          printf("Error opening file\n");
          return;
      }

      double *arr = (double *)malloc(m * sizeof(double));
      if (arr == NULL) {
          fprintf(stderr, "Memory allocation failed\n");
          return;
      }

      double ttime = 0.0;
      clock_t start_time, end_time;

      // Сначала выводим размер массива в файл
      fprintf(f, "ELEMENTS TRY TIME TOTAL RCURSIVE\n");
     printf("\n%d\n", m);

     int i = 0;
      for (i = 0; i < s; i++) {
          unsigned long long int shift_count = 0, recursive = 0;  // Обнуляем
счетчики на каждой итерации
          generation(arr, m);  // Генерация массива
          start_time = clock();

          heap(arr, m, &shift_count, &recursive);  // Сортировка с отслеживанием
количества проходов и обменов

          end_time = clock();

          ttime = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
          fprintf(f, "%d %2d %.8f %llu %llu\n", m, i + 1, ttime, shift_count,
recursive);  // Выводим результаты в файл
          printf("%d ", i + 1);  // Выводим номер текущего прогона на экран
      }

      fclose(f);
      free(arr);  // Освобождаем память
  }
```

- Функция measure
  - Вход: количество элементов, количество попыток в серии.
  - Назначение: Выполняет замеры времени работы сортировки для массивов различных размеров.
  - Как работает: Получаем массив случайных чисел. Для каждого прогона (всего s прогонов) измеряется время выполнения сортировки методом пузырька с использованием clock(). Результаты измерений, включая время, количество проходов и перестановок, записываются в файл times.txt.
  - Результат: Файл times.txt с необходимыми данными.

```
int main() {
    srand(time(NULL)); // Инициализация генератора случайных чисел

    int sizes[] = {1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]), series = 20;

    FILE *f = fopen("times.txt", "a");
    if (!f) {
        printf("Error opening file\n");
        return 1;
    }



   int i = 0;
    for(i = 0; i < num_sizes; i++) {
        measure(sizes[i], series);
    }

    fclose(f);
    return 0;
}
```

- Функция main
    - Назначение: Инициализирует программу, настраивает параметры и вызывает функцию измерения для различных размеров массивов.
    - Как работает: Массив sizes[] содержит размеры массивов для тестирования. Для каждого размера массива вызывается функция measure, которая выполняет сортировку и измеряет время. Результаты всех тестов записываются в файл times.txt.
- Результатом работы программы является файл times.txt, в котором присутствуют:
    - Количество элементов
    - Номер попытки
    - Измеренное время
    - Общее количество вызовов функции
    - Количество рекурсивных вызовов функции

Данная программа реализует алгоритм пирамидальной сортировки и тестирует его производительность на массивах различных размеров. Для каждого массива измеряется время сортировки, количество перестановок элементов и обходов массива.

Заключение.

В ходе выполнения лабораторной работы была реализована пирамидальная сортировка на языке С и приведено ее тестирование на массивах различного размера. Для анализа эффективности заданной сортировки были измерены некоторые параметры времени и количество вызовов функции образования кучи (общих и рекурсивных) (подр. в разделе “Задание”).

Для более наглядного представления на основе полученных данных построены 5 диаграмм:

1. График наихудшего времени выполнения сортировки и сложности алгоритма указанной в нотации О большое.

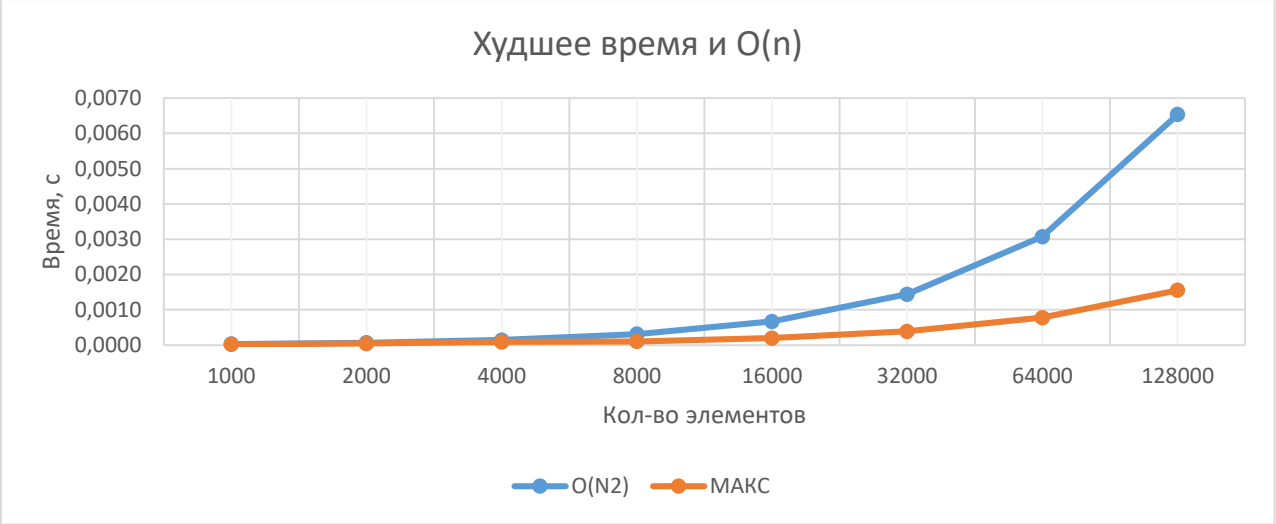

рис. 1. Худшее время и O(nlogn)

2. График среднего, наилучшего и наихудшего времени исполнения.

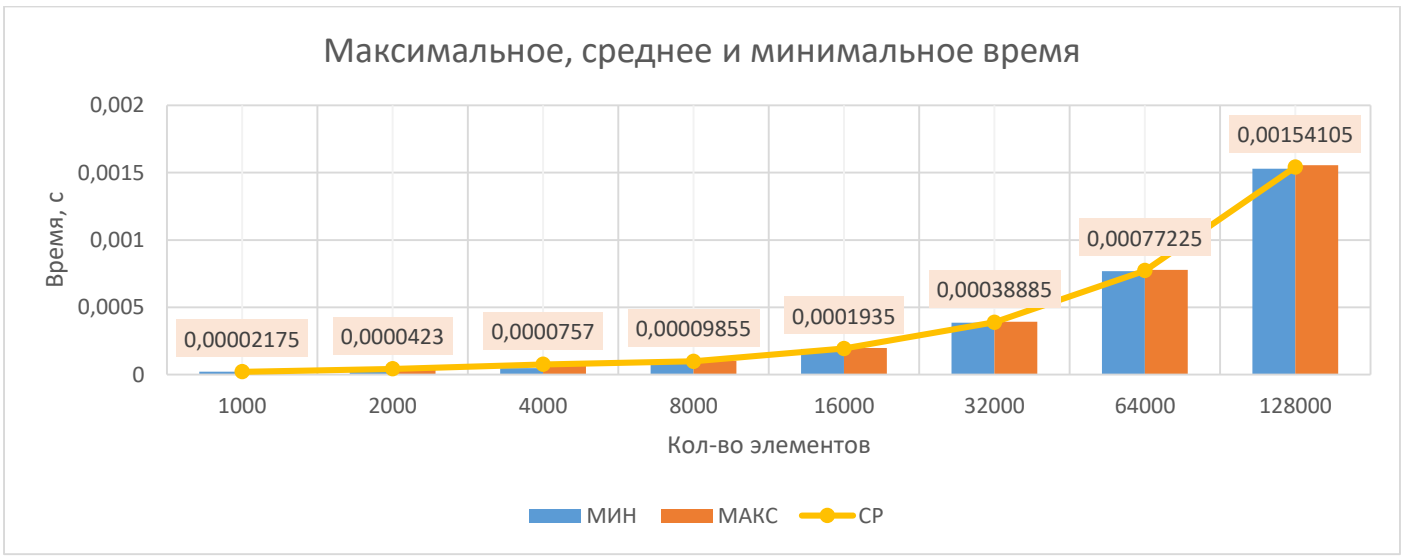

рис. 2. Лучшее, среднее и худшее время исполнения

3. График среднего, наилучшего и наихудшего глубины рекурсии.

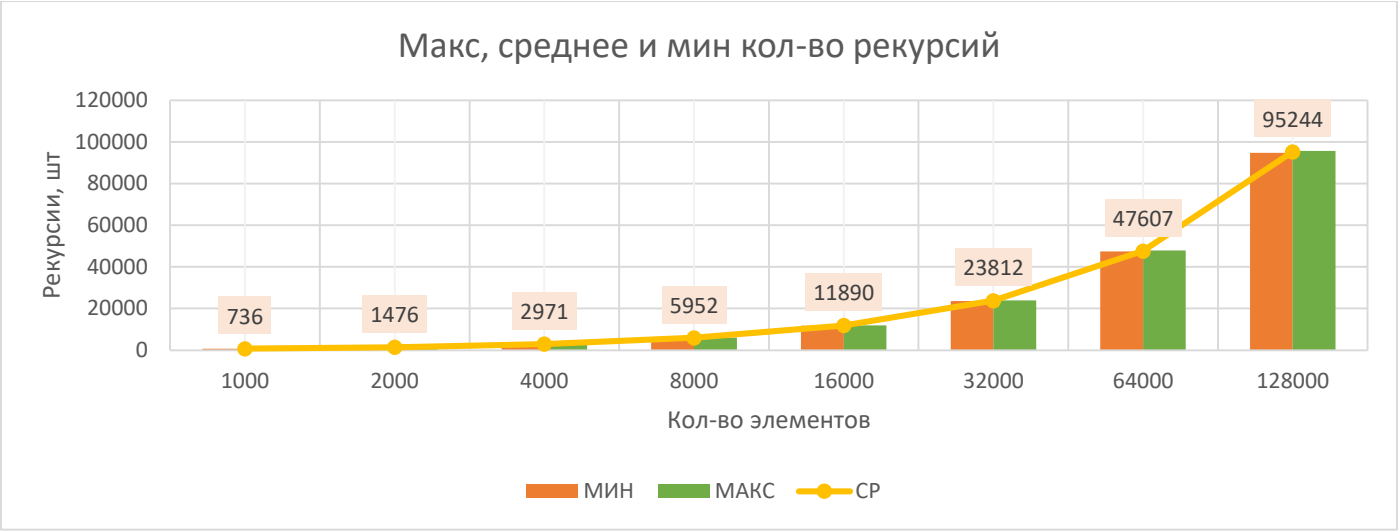

рис. 3. Максимальное, среднее, минимальное число рекурсивных вызовов

4. График среднего по серии количества вызовов функции построения кучи и количества вызовов внутренней функции.

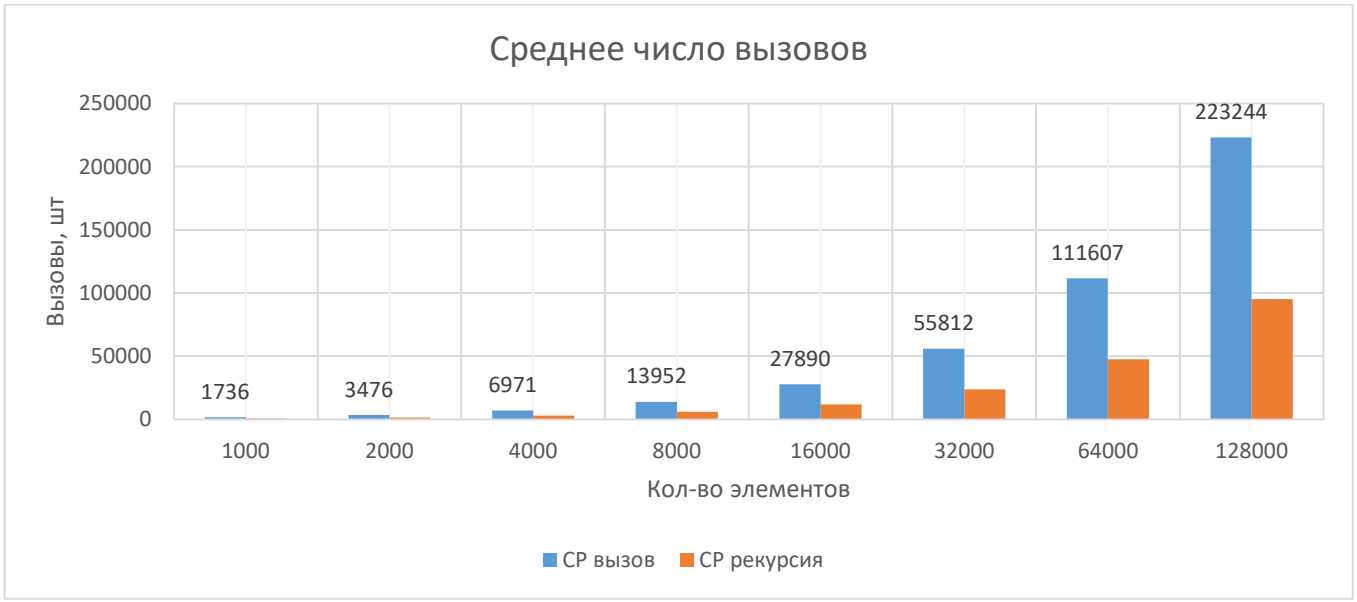

рис. 4. Среднее число общих и рекурсивных вызовов

5. График среднего процентного соотношения вызовов внутренней функции к общему вызову функции.

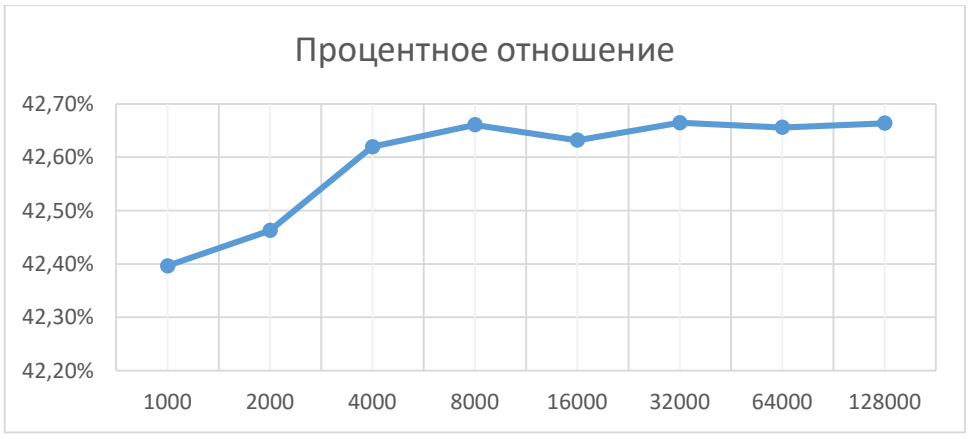

рис. 5. Среднее процентное отношение рекурсивных вызовов к общим

На основании построенных графиков можно подтвердить теоретическую оценку сложности алгоритма пирамидальной сортировки O(n log n). Диаграммы показывают, что время выполнения растет логарифмически с увеличением размера массива, что соответствует ожиданиям для этого алгоритма. Среднее количество вызовов функции образования кучи из массива подтверждают, что алгоритм эффективно использует свойства кучи, выполняя логарифмическое количество операций для каждого извлечения максимального элемента. Также было установлено, что даже при увеличении объема данных пирамидальная сортировка остается относительно быстрой, что делает её более эффективной для больших массивов.