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

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

Выполнил студент группы КС-36: Золотухин А.А.
Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/ZolotukhinAA_36_ALG
Принял: Крашенников Роман Сергеевич
Дата сдачи: 03.03.2025

Москва
2025

Оглавление

Описание задачи 1
Описание метода/модели 2
Выполнение задачи 3
Выводы 10

Описание задачи

В лабораторной работе предлагается изучить альтернативные первой лабораторной работы сортировки, которые обладают меньшей асимптотической сложностью и сравнить их с результатами предыдущей лабораторной работы.

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

Задание:

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

  За один расчёт выполняются следующие операции:

  1. Генерируется массив случайных значений;
  2. Запоминается время начала расчёта алгоритма сортировки;
  3. Выполняется алгоритм сортировки
  4. Вычисляется время, затраченное на сортировку: текущее время - время начала;
  5. Сохраняется время для одной попытки.

     После этого расчёт 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 значениями чисел с плавающей точкой в интервале от -1 до 1;
     - Для серии запоминаются все времена, которые были замерены.

- По полученным данным времени построить графики зависимости времени от числа элементов в массиве:

  1. Совмещенный график наихудшего времени выполнения сортировки и сложности алгоритма, указанной в нотации O большое;

     Для построения графика вычисляется O большое для каждого размера массива. При этом при вычислении функции O(c * g(N)) подбирается такая константа c, чтобы при значении >1000 график O(N) был выше графика наихудшего случая, но второй график на его фоне не превращался в прямую линию.
  2. Совмещенный график среднего, наихудшего и наилучшего времени исполнения;

3. Совмещённый график средней, наилучшей и наихудшей глубины рекурсии;
4. Совмещённый график среднего по серии количество вызовов функции построения кучи и количества вызовов внутренней функции;
5. График среднего процентного соотношения вызовов внутренней функции к общему вызову функции.

- По результатам расчётов оформляется отчёт по предоставленной форме, в отчете:

1. Приводится описание алгоритма;
2. Приводится описание выполнения задачи (описание кода и специфических элементов реализации);
3. Приводятся выводы (Графики и их анализ). Требуется ответить на вопрос о поведении алгоритма, изученного в процессе выполнения лабораторной работы и зафиксировать его особенности.

Описание метода/модели

Пирамидальная сортировка (или, Сортировка кучей) - это метод сортировки на основе сравнения, основанный на двоичной куче данных. При сортировке кучей мы используем двоичную кучу, чтобы быстро находить и перемещать максимальный элемент за *O(logN)* вместо *O(N)* и, следовательно, достигать временной сложности *O(NlogN)*. Ход алгоритма:

1. Переставляем элементы массива так, чтобы они образовывали максимальную кучу;
2. Повторяем следующие шаги до тех пор, пока куча не будет содержать только один элемент:
   (a) Меняем местами корневой элемент кучи с последним элементом кучи;
   (b) Удаляем последний элемент кучи;
   (c) Складываем в кучу остальные элементы кучи.
3. Получаем отсортированный массив.

Анализ сложности пирамидальной сортировки:

- **Лучший вариант: O(N)**, если массив состоит из идентичных элементов;
- **Средний вариант: O($N(logN)$)**, если массив упорядочен случайным образом;
- **Наихудший вариант: O($N(logN)$)**, если массив находится в обратном порядке, где N - количество элементов в массиве.

Преимущества:

- Эффективная временная сложность;
- Использование памяти может быть минимальным;
- Простота.

Недостатки:

- Дорогостоящая, так как константы выше по сравнению с сортировкой слиянием;
- Неэффективен из-за высоких констант во временной сложности.

Выполнение задачи

Алгоритм пирамидальной сортировки реализован на языке *C++*. Построение графиков проводить с помощью программы *GNUplot*.

"main" функция работает с циклом, в ходе которого производится расчёт минимального, максимального и среднего времени на сортировку массива размером M. Каждая серия просчитывается по 20 раз. В итоге получаются данные, выведенные в определенные файлы, с помощью которых впоследствии строятся графики.

```
1    int main() {
2        std::ofstream worst_and_complexity(Constants::folder + "worst_and_complexity.
   dat");
3        std::ofstream average_best_worst(Constants::folder + "average_best_worst.dat"
   );
4        std::ofstream recursion_depth(Constants::folder + "recursion_depth.dat");
5        std::ofstream heap_calls(Constants::folder + "heap_calls.dat");
6        std::ofstream inner_heap_ratio(Constants::folder + "inner_heap_ratio.dat");
7
8        if (!worst_and_complexity.is_open() ||
9      !average_best_worst.is_open() ||
10     !recursion_depth.is_open() ||
11     !heap_calls.is_open() ||
12     !inner_heap_ratio.is_open()) {
13     std::cerr << "Error of opening file!" << std::endl;
14     return 1;
15       }
16
17       for (int episode = 0; episode < Constants::M; episode++) {
18     int size = Constants::sizes[episode];
19     double* array = new double[size];
20
21     long double the_worst_time = 0.0;
22     long double the_best_time = std::numeric_limits<long double>::max();
23     long double total_time = 0.0;
24
25     long long total_heap_calls_all = 0;
26     long long total_inner_heap_calls_all = 0;
27     long long max_recursion_depth_all = 0;
28
```

```
    long double total_recursion_depth = 0.0;
    long double best_recursion_depth = std::numeric_limits<long double>::max();
    long double worst_recursion_depth = 0.0;

    for (int attempt = 0; attempt < Constants::amount_of_attempts; attempt++) {
        long long total_heap_calls = 0;
        long long total_inner_heap_calls = 0;
        long long current_depth = 0;
        long long max_recursion_depth = 0;

        generationArray(array, size);

        std::chrono::high_resolution_clock::time_point start = std::chrono::high_resolution_clock::now();

        heapSort(array, size, total_heap_calls, total_inner_heap_calls, max_recursion_depth);

        std::chrono::high_resolution_clock::time_point end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<long double, std::milli> milli_diff = end - start;

        long double time_taken = milli_diff.count();
        if (time_taken > the_worst_time)
      the_worst_time = time_taken;
        if (time_taken < the_best_time)
      the_best_time = time_taken;

        total_time += time_taken;
        total_heap_calls_all += total_heap_calls;
        total_inner_heap_calls_all += total_inner_heap_calls;

        if (max_recursion_depth > max_recursion_depth_all)
      max_recursion_depth_all = max_recursion_depth;

        total_recursion_depth += max_recursion_depth;
        if (max_recursion_depth < best_recursion_depth)
      best_recursion_depth = max_recursion_depth;
        if (max_recursion_depth > worst_recursion_depth)
      worst_recursion_depth = max_recursion_depth;

        std::cout << "Time: " << time_taken << " ms." << std::endl;
    }

    long double average_time = total_time / Constants::amount_of_attempts;
    long double complexity = Constants::c * static_cast<long double>(size) * std::log(size);
    std::cout << complexity << std::endl;

    long double average_heap_calls = static_cast<long double>(total_heap_calls_all) / Constants::amount_of_attempts;
    long double average_inner_heap_calls = static_cast<long double>(total_inner_heap_calls_all) / Constants::amount_of_attempts;
    long double inner_heap_ratio_value = (average_inner_heap_calls / average_heap_calls) * 100.0;

    long double average_recursion_depth = total_recursion_depth / Constants::amount_of_attempts;

```

```
80      worst_and_complexity << size << " " << the_worst_time << " " << complexity <<
    std::endl;
81      average_best_worst << size << " " << average_time << " " << the_best_time << "
    " << the_worst_time << std::endl;
82      recursion_depth << size << " " << average_recursion_depth << " " <<
    best_recursion_depth << " " << worst_recursion_depth << std::endl;
83      heap_calls << size << " " << average_heap_calls << " " <<
    average_inner_heap_calls << std::endl;
84      inner_heap_ratio << size << " " << inner_heap_ratio_value << std::endl;
85
86      delete[] array;
87        }
88
89        worst_and_complexity.close();
90        average_best_worst.close();
91        recursion_depth.close();
92        heap_calls.close();
93        inner_heap_ratio.close();
94
95        return 0;
96    }
97
```

"generationArray" функция принимает два аргумента: array - массив, size - размер массива. Формирует массив размера size, который заполняется случайными числами с плавающей точкой от -1 до 1.

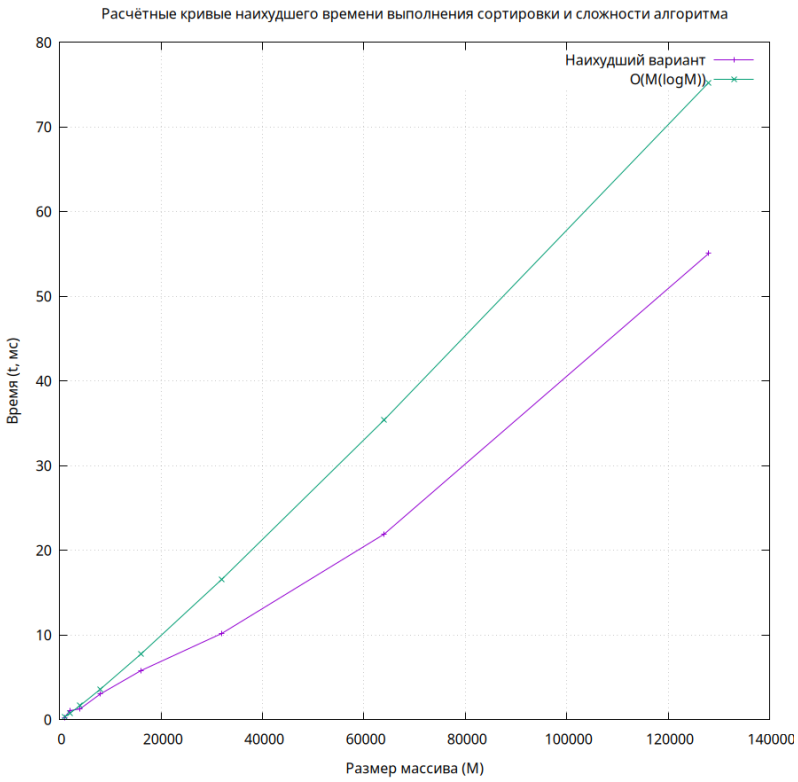
```
1     void generationArray(double array[], int size) {
2       std::random_device rd;
3       std::mt19937 engine(rd());
4       std::uniform_real_distribution<double> gen(-1.0, 1.0);
5
6       for (int i = 0; i < size; i++)
7         array[i] = gen(engine);
8     }
9
```

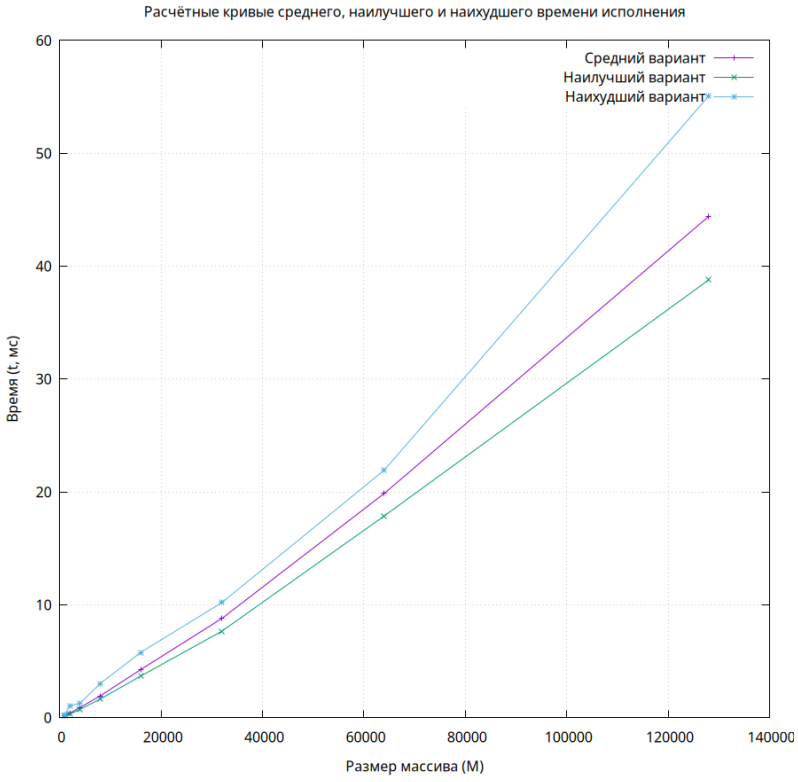
"heapSort" функция принимает пять аргументов: array - массив, size - размер массива, total_heap_calls - общее число вызовов кучи, total_inner_heap_calls - общее число внутренних вызовов кучи, max_recursion_depth - максимальная глубина рекурсии. Сортирует массив пирамидальным методом и просчитывает общее число вызовов кучи, общее число внутренних вызовов кучи и максимальную глубину рекурсии.

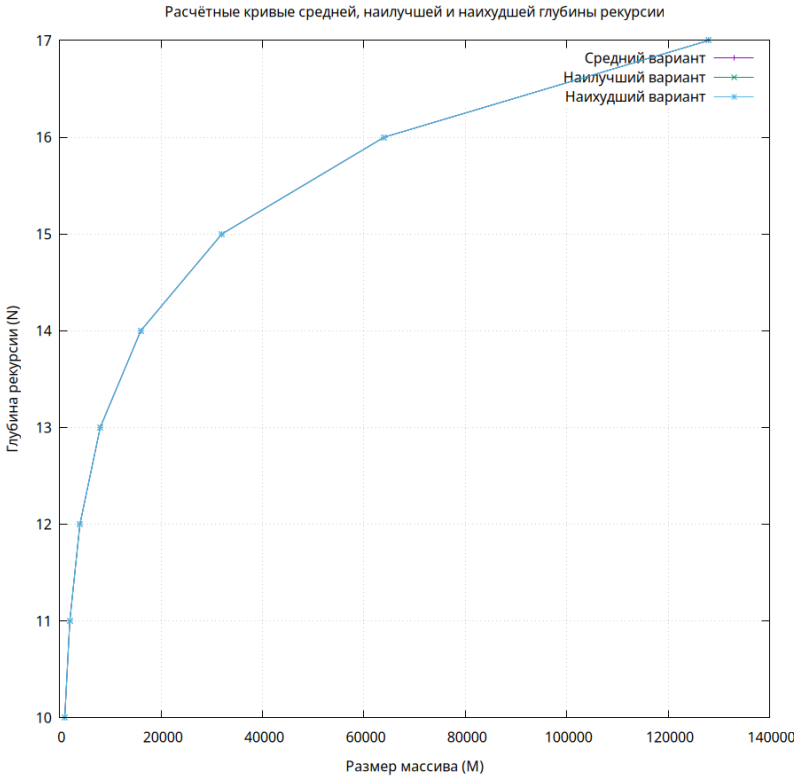
```
1     void heapSort(double array[], int size, long long& total_heap_calls, long long&
    total_inner_heap_calls, long long& max_recursion_depth) {
2       for (int i = (size / 2 - 1); i >= 0; i--) {
3       long long current_depth = 0;
4       makeHeap(array, size, i, total_heap_calls, total_inner_heap_calls,
    current_depth, max_recursion_depth);
5       }
6
7       for (int i = (size - 1); i >= 0; i--) {
8       std::swap(array[0], array[i]);
9       long long current_depth = 0;
10      makeHeap(array, i, 0, total_heap_calls, total_inner_heap_calls, current_depth,
    max_recursion_depth);
11      }
12    }
```

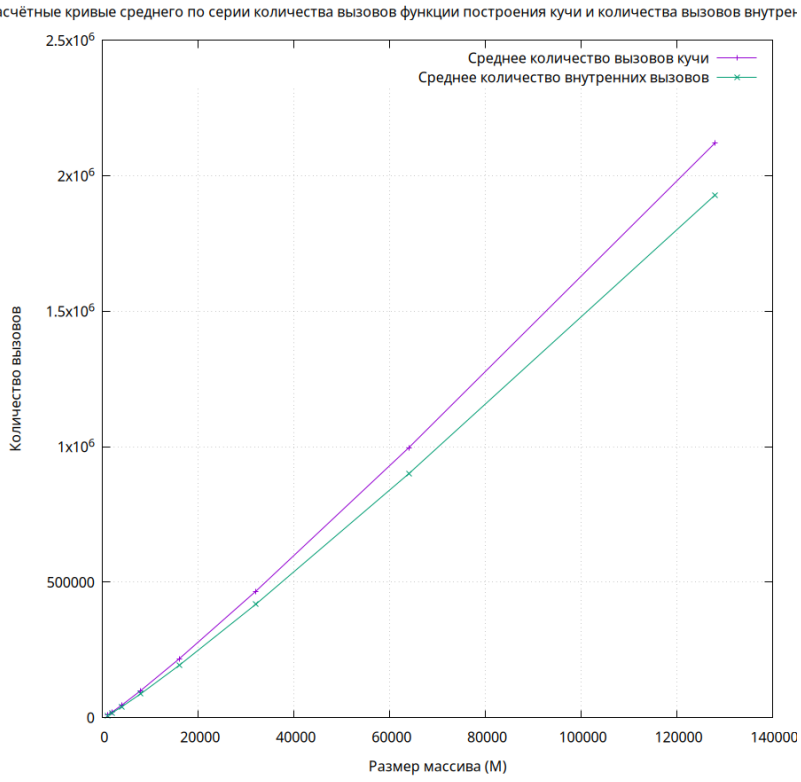
"makeHeap" функция принимает 6 аргументов: array - массив, size - размер массива, i - текущий индекс, total_heap_calls - общее число вызовов кучи, total_inner_heap_calls - общее число внутренних вызовов кучи, max_recursion_depth - максимальная глубина рекурсии. Создаёт "максимальную"кучу для неупорядоченного массива и просчитывает общее число вызовов кучи, общее число внутренних вызовов кучи и максимальную глубину рекурсии.

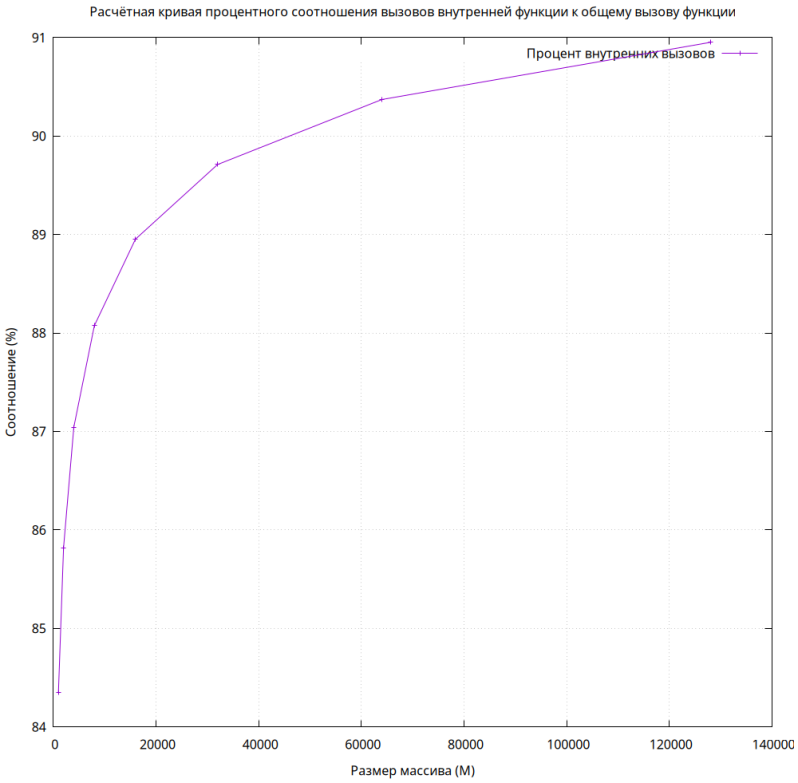
```
 void makeHeap(double array[], int size, int i, long long& total_heap_calls, long long& total_inner_heap_calls, long long& current_depth, long long& max_recursion_depth) {
     total_heap_calls++;
     current_depth++;

     if (current_depth > max_recursion_depth)
   max_recursion_depth = current_depth;

     int largest = i;
     int l = 2 * i + 1;
     int r = 2 * i + 2;

     if (l < size && array[l] > array[largest])
   largest = l;

     if (r < size && array[r] > array[largest])
   largest = r;

     if (largest != i) {
   std::swap(array[i], array[largest]);
   makeHeap(array, size, largest, total_heap_calls, total_inner_heap_calls, current_depth, max_recursion_depth);
   total_inner_heap_calls++;
     }
 }
```


Расчётные кривые наихудшего времени выполнения сортировки и сложности алгоритма
Наихудший вариант
O(M(logM))
Время (t, мс)
Размер массива (M)
0
10
20
30
40
50
60
70
80
0
20000
40000
60000
80000
100000
120000
140000


Расчётные кривые среднего, наилучшего и наихудшего времени исполнения
Средний вариант
Наилучший вариант
Наихудший вариант
Время (t, мс)
Размер массива (M)
0
10
20
30
40
50
60
0
20000
40000
60000
80000
100000
120000
140000


Расчётные кривые средней, наилучшей и наихудшей глубины рекурсии
Средний вариант
Наилучший вариант
Наихудший вариант
Глубина рекурсии (N)
Размер массива (M)
10
11
12
13
14
15
16
17
0
20000
40000
60000
80000
100000
120000
140000


асчётные кривые среднего по серии количества вызовов функции построения кучи и количества вызовов внутрен
Среднее количество вызовов кучи
Среднее количество внутренних вызовов
Количество вызовов
Размер массива (M)
0
500000
1x10^6
1.5x10^6
2x10^6
2.5x10^6
0
20000
40000
60000
80000
100000
120000
140000


Расчётная кривая процентного соотношения вызовов внутренней функции к общему вызову функции
Процент внутренних вызовов
Соотношение (%)
Размер массива (M)
91
90
89
88
87
86
85
84
0
20000
40000
60000
80000
100000
120000
140000

Выводы

Сортировка состоит из двух циклов, которые последовательны со сложностью *O(NlogN)*.

Касательно производительности, алгоритм на практике проигрывает по скорости как быстрой, так и сортировке слиянием, и так же как и сортировка слиянием алгоритм не работает быстрее на частично отсортированных данных, поэтому его часто модифицируют. Поэтому на больших массивах (до нескольких тысяч) медленне сортировки Шелла со сложностью $O(N^2)$, но в лучшем случае $O(Nlog^2N)$.

Из важных достоинств также является отсутствие потребления памяти.

По сравнению с сортировкой вставками, данная сортировка гарантированно сортирует массив с временной сложностью $O(Nlog^2N)$, когда первая сортирует со сложностью $O(N^2)$ (или $O(N)$ влучшем случае). Быстрее данная сортировка за счёт использования бинарного дерева, в котором поиск и удаления элементов происходит за $O(logN)$.