Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

№ 2 ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»:

СТУДЕНТ группы КС-33 Костяева К.С.

Москва
2024

ОГЛАВЛЕНИЕ

ТЕОРИЯ

Пирамидальная сортировка (или сортировка кучей, HeapSort) — это метод сортировки сравнением, основанный на такой структуре данных как двоичная куча. Она похожа на сортировку выбором, где мы сначала ищем максимальный элемент и помещаем его в конец. Далее мы повторяем ту же операцию для оставшихся элементов.

Алгоритм:

1. Построение кучи:

- Начинаем с того, что считаем, что все элементы массива организованы в виде бинарного дерева, не удовлетворяющего свойству кучи.
- Для этого нам нужно пройти по всем узлам, начиная с последнего родительского узла (то есть узла, который не имеет потомков) и применить операцию **heapify** для всех узлов, чтобы превратить дерево в кучу.

2. Сортировка:

- После того как куча построена, максимальный элемент (находящийся в корне) перемещается в конец массива.
- Далее уменьшаем размер кучи на 1, а затем восстанавливаем кучу, применяя **heapify** к корню дерева, чтобы максимальный элемент снова оказался в корне.
- Этот процесс повторяется до тех пор, пока весь массив не будет отсортирован.

3. Пример:

```
Входные данные: 4, 10, 3, 5, 1
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)
Числа в скобках представляют индексы в представлении данных в виде массива.
Применение процедуры heapify к индексу 1:
         4(0)
        /   \
    10(1)    3(2)
    /   \
5(3)    1(4)
Применение процедуры heapify к индексу 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)
Процедура heapify вызывает себя рекурсивно для создания кучи  сверху вниз.
```

1. Построение кучи (Heapify)

Нам нужно преобразовать массив в **максимальную кучу** (где каждый родитель больше потомков).

Шаг 1: Работаем с поддеревом, где корень — 10 (индекс 1)

- Потомки: **5 (индекс 3)** и **1 (индекс 4)**
- **10 уже больше обоих**, ничего не меняем.

Шаг 2: Работаем с поддеревом, где корень — 4 (индекс 0)

- Потомки: **10 (индекс 1)** и **3 (индекс 2)**
- **10 больше 4**, меняем их местами:
  [10, 4, 3, 5, 1]
- Теперь у 4 потомки **5 (индекс 3)** и **1 (индекс 4)**
- **5 больше 4**, меняем их местами:
  [10, 5, 3, 4, 1]
- Получили максимальную кучу.

2. Сортировка (извлечение элементов)

Теперь будем **по очереди ставить максимальный элемент в конец массива**.

Шаг 1: Меняем корень (10) с последним элементом (1)

[1, 5, 3, 4, 10], теперь 10 на своем месте.
Восстанавливаем кучу на `[1, 5, 3, 4]`

- **5 — новый корень**, больше 1 → меняем их:
  [5, 1, 3, 4, 10]
- Теперь у 1 потомки **4** → 4 больше → меняем:
  [5, 4, 3, 1, 10]

Шаг 2: Меняем корень (5) с последним неотсортированным (1)

[1, 4, 3, 5, 10], 5 встает на место.
Восстанавливаем кучу на `[1, 4, 3]`

- **4 — новый корень**, больше 1 → меняем:
  [4, 1, 3, 5, 10]

Шаг 3: Меняем корень (4) с 3

[3, 1, 4, 5, 10], 4 на месте.
Осталась мини-куча `[3, 1]` → 3 больше, остается.

Шаг 4: Меняем 3 с 1

[1, 3, 4, 5, 10]

Итог:

[1, 3, 4, 5, 10] — массив отсортирован

Время работы:

1. Время построения кучи:

Построение кучи занимает O(n), так как каждый вызов `heapify` обрабатывает элементы в глубину, и для всех элементов требуется O(n) времени.

2. Время сортировки:

На каждом шаге извлечения максимального элемента и восстановления кучи потребуется O(log(n)). Мы выполняем эту операцию n−1 раз, следовательно, общая сложность сортировки — O(n*log(n)).

Таким образом, общая сложность алгоритма пирамидальной сортировки составляет **O(n*log(n)).**

ЗАДАНИЕ

Вариант 3

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. Вполняется 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:

  - общее колличество вызовов функции построения кучи
  - колличество вызовов внутренней функции построения кучи(вызов внутри функции формирования кучи)
  - времени исполнения сортировки

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

  i. Совмешееый график наихудшего времени выполнения сортировки и сложности алгоритма указанной в нотации О большое.
  > Для построения графика вычисляется О большое для каждого размера массива. При этом при вычислении функции O(c * g(N)) подбирается такая константа c, что бы при значении > 1000 график O(N) был выше графика наихудшего случая, но второй график на его фоне не превращался в прямую линию

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

ПРАКТИЧЕСКАЯ ЧАСТЬ

Код:

Функция heapify:

язык программирования C++

```
// Функция построения кучи (heapify)
void heapify(vector<double>& arr, int n, int i, int depth) {
    heapifyCalls++;
    recursionDepth = max(recursionDepth, (long long)depth);  // Подсчет глубины
рекурсии
    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) {
        swap(arr[i], arr[largest]);
        internalHeapifyCalls++;
        heapify(arr, n, largest, depth + 1);
    }
}
```

Эта функция рекурсивно преобразует подмассив в кучу (heap). Она принимает вектор arr, размер кучи n, индекс текущего элемента i и текущую глубину рекурсии depth.

В ходе выполнения функции вычисляется, является ли текущий элемент наибольшим среди его поддерева. Если нет, то выполняется обмен значений и происходит дальнейшее рекурсивное преобразование поддерева.

В функции также ведется учет числа вызовов и глубины рекурсии.

Функция heapSort:

язык программирования C++

```
// Пирамидальная сортировка
void heapSort(vector<double>& arr) {
    int n = arr.size();

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i, 1);  // Инициализируем глубину рекурсии с 1
    }

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

Это основная функция для пирамидальной сортировки. Сначала она строит кучу для всего массива, начиная с последнего родительского узла, а затем извлекает элементы из кучи, повторяя процесс сортировки.

В процессе работы отслеживаются вызовы heapify для построения кучи и сортировки.

Функция generateRandomArray:

язык программирования C++

```
// Генерация массива случайных чисел с плавающей запятой
void generateRandomArray(vector<double>& arr, int size, mt19937& engine,
uniform_real_distribution<double>& gen) {
    for (auto& el : arr) {
        el = gen(engine);
    }
}
```

Генерирует массив случайных чисел с плавающей запятой в заданном диапазоне `[-1.0, 1.0]`.

Функция runTests:

язык программирования C++

```
// Функция для проведения серии тестов
void runTests(vector<int>& sizes, int seriesCount) {
    mt19937 engine(time(0));
    uniform_real_distribution<double> gen(-1.0, 1.0);

    ofstream outputFile("heap_sort_results.csv");  // Открываем файл для записи

    // Заголовок для CSV
    outputFile << "Size, Average Time, Min Time, Max Time, Avg Heapify Calls, Avg
Internal Heapify Calls, Max Recursion Depth, Avg Recursion Depth, Heapify Calls
Percentage\n";

    for (int size : sizes) {
        vector<double> times;
        vector<long long> heapifyCounts;
        vector<long long> internalHeapifyCounts;
        vector<long long> maxRecursionDepths;
        vector<long long> avgRecursionDepths;

        for (int i = 0; i < seriesCount; ++i) {
            vector<double> arr(size);
            generateRandomArray(arr, size, engine, gen);

            heapifyCalls = 0;
            internalHeapifyCalls = 0;
            recursionDepth = 0;

            auto start = high_resolution_clock::now();
            heapSort(arr);
            auto end = high_resolution_clock::now();

            duration<double> diff = end - start;
            times.push_back(diff.count());
            heapifyCounts.push_back(heapifyCalls);
            internalHeapifyCounts.push_back(internalHeapifyCalls);
            maxRecursionDepths.push_back(recursionDepth);
            avgRecursionDepths.push_back(recursionDepth / (long long)seriesCount);
// Усредненная глубина
        }

        // Вычисляем необходимые статистики
        double minTime = *min_element(times.begin(), times.end());
        double maxTime = *max_element(times.begin(), times.end());
        double avgTime = accumulate(times.begin(), times.end(), 0.0) / seriesCount;
        long long avgHeapifyCalls = accumulate(heapifyCounts.begin(),
heapifyCounts.end(), 0LL) / seriesCount;
        long long avgInternalHeapifyCalls = accumulate(internalHeapifyCounts.begin(),
internalHeapifyCounts.end(), 0LL) / seriesCount;
```

```
        long long maxRecursionDepth = *max_element(maxRecursionDepths.begin(),
maxRecursionDepths.end());
        long long avgRecursionDepth = accumulate(avgRecursionDepths.begin(),
avgRecursionDepths.end(), 0LL) / seriesCount;

        // Записываем данные в CSV
        outputFile << size << ", " << avgTime << ", " << minTime << ", " << maxTime
<< ", "
            << avgHeapifyCalls << ", " << avgInternalHeapifyCalls << ", "
            << maxRecursionDepth << ", " << avgRecursionDepth << ", "
            << (double)internalHeapifyCalls / heapifyCalls * 100 << "\n";

    outputFile.close();  // Закрываем файл
}
```

Эта функция проводит серию тестов на различных размерах массивов. Для каждого размера она выполняет несколько прогонов сортировки, чтобы собрать статистику времени выполнения и другие параметры.

Для каждого размера массива собираются:

- Время выполнения сортировки (минимальное, максимальное, среднее).
- Количество вызовов функции `heapify`.
- Максимальная глубина рекурсии.
- Средняя глубина рекурсии.
- Процент внутренних вызовов функции `heapify` (показатель, насколько часто требуется выполнять перестановку элементов).

Результаты выводятся на экран и записываются в CSV-файл для дальнейшего анализа.

Основная функция main:

язык программирования C++

```
int main() {
    vector<int> sizes = { 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000 };
    int seriesCount = 20;

    runTests(sizes, seriesCount);

    return 0;
}
```

В основной функции задаются размеры массивов для тестирования и количество прогонов для каждого размера.

Вызывается функция `runTests` для выполнения всех тестов.

Результат работы программы:

вывод в файл CSV

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Size, Average Time, Min Time, Max Time, Avg Heapify Calls, Avg Internal Heapify Calls, Max Recursion Depth, Avg Recursion Depth, Heapify Calls Percentage														
2	1000, 0.00035377, 0.0002899, 0.000463, 9581, 8082, 10, 0, 84.3316														
3	2000, 0.00070637, 0.0006449, 0.0010588, 21157, 18158, 11, 0, 85.831														
4	4000, 0.00150879, 0.0014143, 0.0019903, 46326, 40327, 12, 0, 87.0354														
5	8000, 0.0032452, 0.0030042, 0.0039401, 100626, 88627, 13, 0, 88.0882														
6	16000, 0.00703548, 0.0066298, 0.007583, 217252, 193253, 14, 0, 88.9549														
7	32000, 0.0159657, 0.0145044, 0.0205116, 466499, 418500, 15, 0, 89.7111														
8	64000, 0.0318204, 0.0299099, 0.0346359, 996990, 900991, 16, 0, 90.3717														
9	128000, 0.071051, 0.0667544, 0.0781445, 2121953, 1929954, 17, 0, 90.9527														
10															
11															

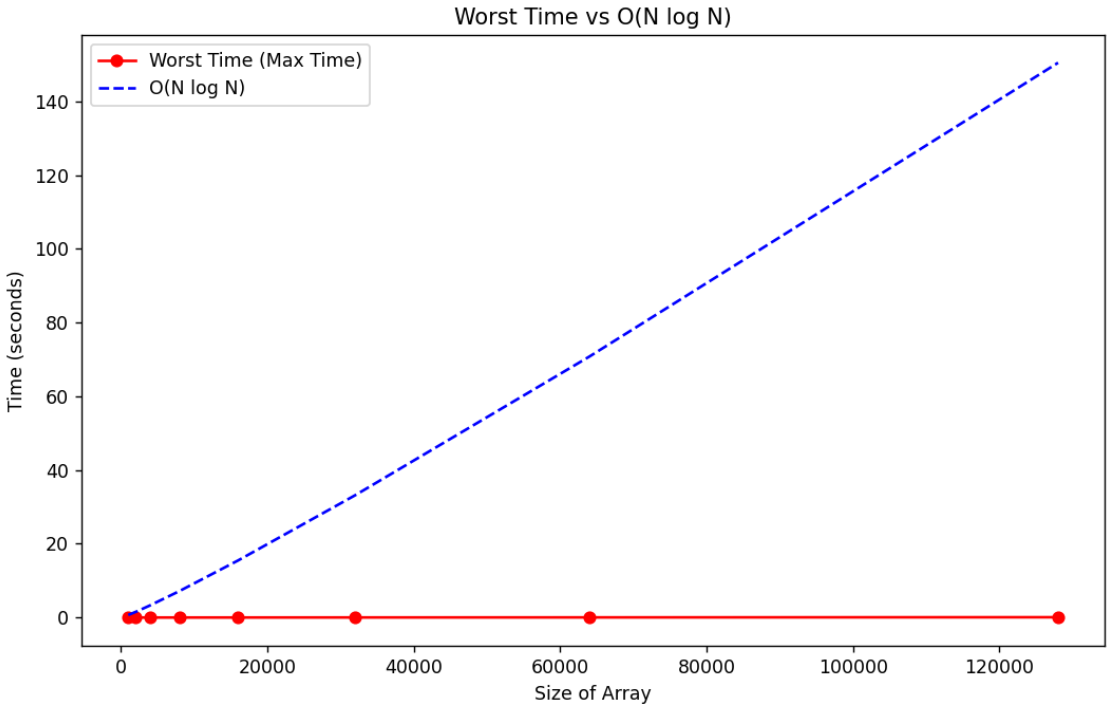
вывод в консоль

```
Size: 1000
  Average Time: 0.00035377 sec
  Min Time: 0.0002899 sec
  Max Time: 0.000463 sec
  Avg Heapify Calls: 9581
  Avg Internal Heapify Calls: 8082
  Max Recursion Depth: 10
  Avg Recursion Depth: 0
  Heapify Calls Percentage: 84.3316%
Size: 2000
  Average Time: 0.00070637 sec
  Min Time: 0.0006449 sec
  Max Time: 0.0010588 sec
  Avg Heapify Calls: 21157
  Avg Internal Heapify Calls: 18158
  Max Recursion Depth: 11
  Avg Recursion Depth: 0
  Heapify Calls Percentage: 85.831%
Size: 4000
  Average Time: 0.00150879 sec
  Min Time: 0.0014143 sec
  Max Time: 0.0019903 sec
  Avg Heapify Calls: 46326
  Avg Internal Heapify Calls: 40327
  Max Recursion Depth: 12
  Avg Recursion Depth: 0
  Heapify Calls Percentage: 87.0354%
Size: 8000
  Average Time: 0.0032452 sec
  Min Time: 0.0030042 sec
  Max Time: 0.0039401 sec
  Avg Heapify Calls: 100626
  Avg Internal Heapify Calls: 88627
  Max Recursion Depth: 13
  Avg Recursion Depth: 0
  Heapify Calls Percentage: 88.0882%
```

```
Size: 16000
  Average Time: 0.00703548 sec
  Min Time: 0.0066298 sec
  Max Time: 0.007583 sec
  Avg Heapify Calls: 217252
  Avg Internal Heapify Calls: 193253
  Max Recursion Depth: 14
  Avg Recursion Depth: 0
  Heapify Calls Percentage: 88.9549%
Size: 32000
  Average Time: 0.0159657 sec
  Min Time: 0.0145044 sec
  Max Time: 0.0205116 sec
  Avg Heapify Calls: 466499
  Avg Internal Heapify Calls: 418500
  Max Recursion Depth: 15
  Avg Recursion Depth: 0
  Heapify Calls Percentage: 89.7111%
Size: 64000
  Average Time: 0.0318204 sec
  Min Time: 0.0299099 sec
  Max Time: 0.0346359 sec
  Avg Heapify Calls: 996990
  Avg Internal Heapify Calls: 900991
  Max Recursion Depth: 16
  Avg Recursion Depth: 0
  Heapify Calls Percentage: 90.3717%
Size: 128000
  Average Time: 0.071051 sec
  Min Time: 0.0667544 sec
  Max Time: 0.0781445 sec
  Avg Heapify Calls: 2121953
  Avg Internal Heapify Calls: 1929954
  Max Recursion Depth: 17
  Avg Recursion Depth: 0
  Heapify Calls Percentage: 90.9527%
```

Графики:

1. График наихудшего времени выполнения сортировки и O(N log N)

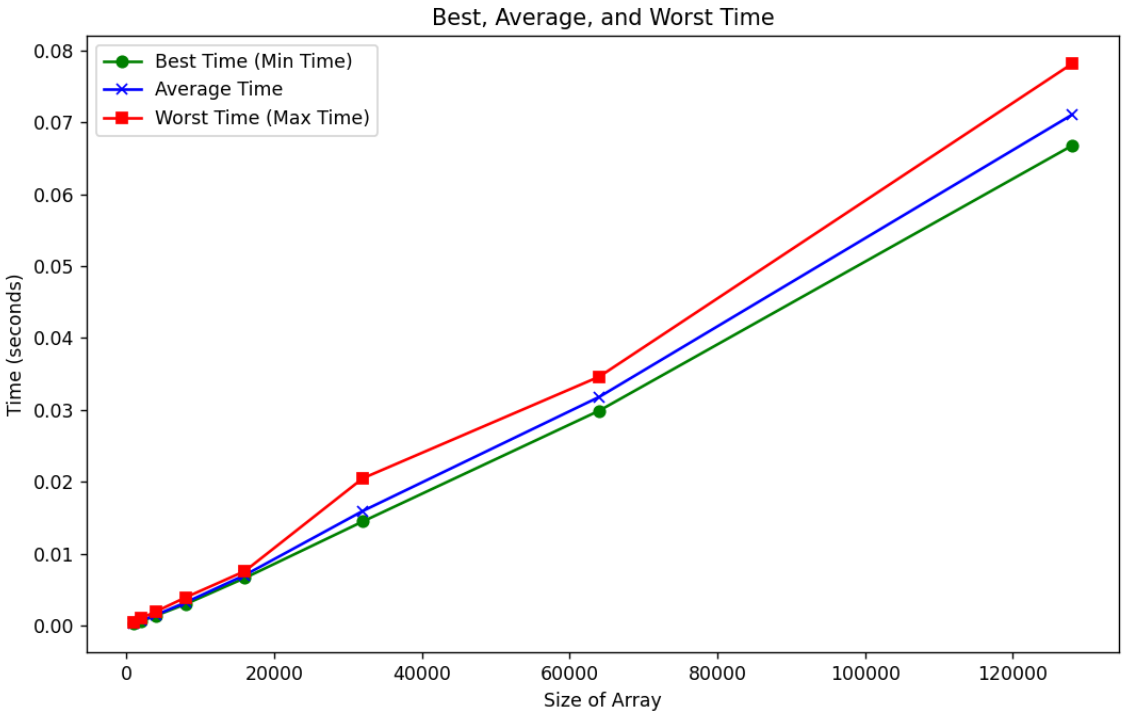


Красная линия (Worst Time): Это реальное время, которое потребовалось для сортировки с наибольшим временем.

Синяя пунктирная линия (O(N log N)): Это теоретическая линия, которая демонстрирует, как должно расти время сортировки, если бы оно зависело только от сложности алгоритма.

Этот график сравнивает реальные данные (максимальное время выполнения сортировки) с теоретической оценкой сложности O(N log N). В идеале время наихудшего случая должно следовать подобной кривой, что подтверждает правильность оценки сложности алгоритма.

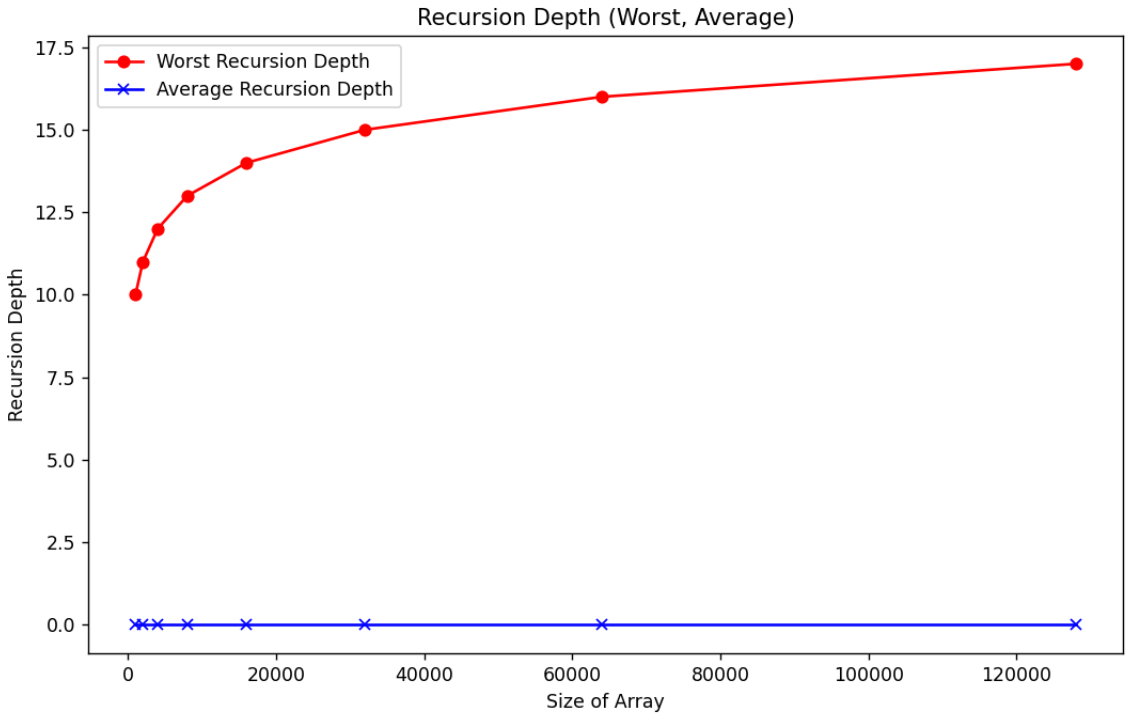
2. График среднего, наилучшего и наихудшего времени исполнения



Зеленая линия (Min Time): Это минимальное время сортировки, которое может быть получено.
Синяя линия (Average Time): Среднее время выполнения сортировки.
Красная линия (Max Time): Время сортировки в наихудшем случае.

Этот график показывает, как различные типы времени (минимальное, среднее и максимальное) изменяются в зависимости от размера массива. Он позволяет увидеть, как "плавающее" время может варьироваться в зависимости от случайности входных данных.

3. График глубины рекурсии (лучший и наихудший случаи)

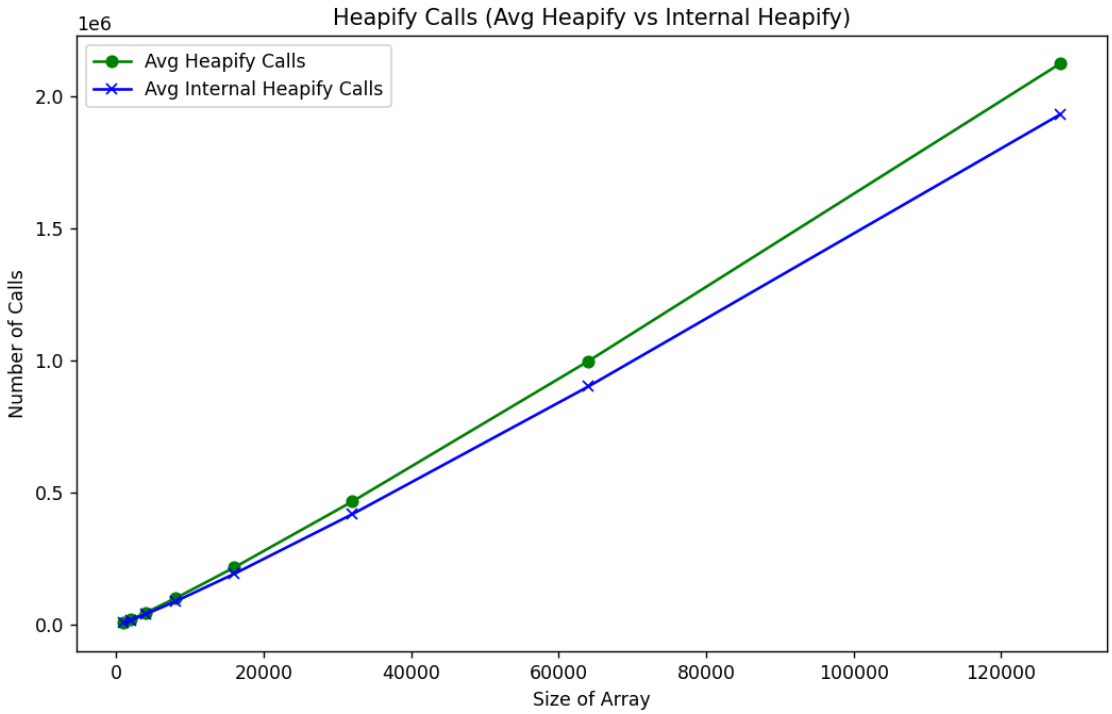


Красная линия (Worst Recursion Depth): Это максимальная глубина рекурсии в тестах для каждого размера массива.

Синяя линия (Average Recursion Depth): Средняя глубина рекурсии по всем тестам.

График помогает проанализировать, как глубина рекурсии меняется в зависимости от размера массива и как она влияет на производительность алгоритма.

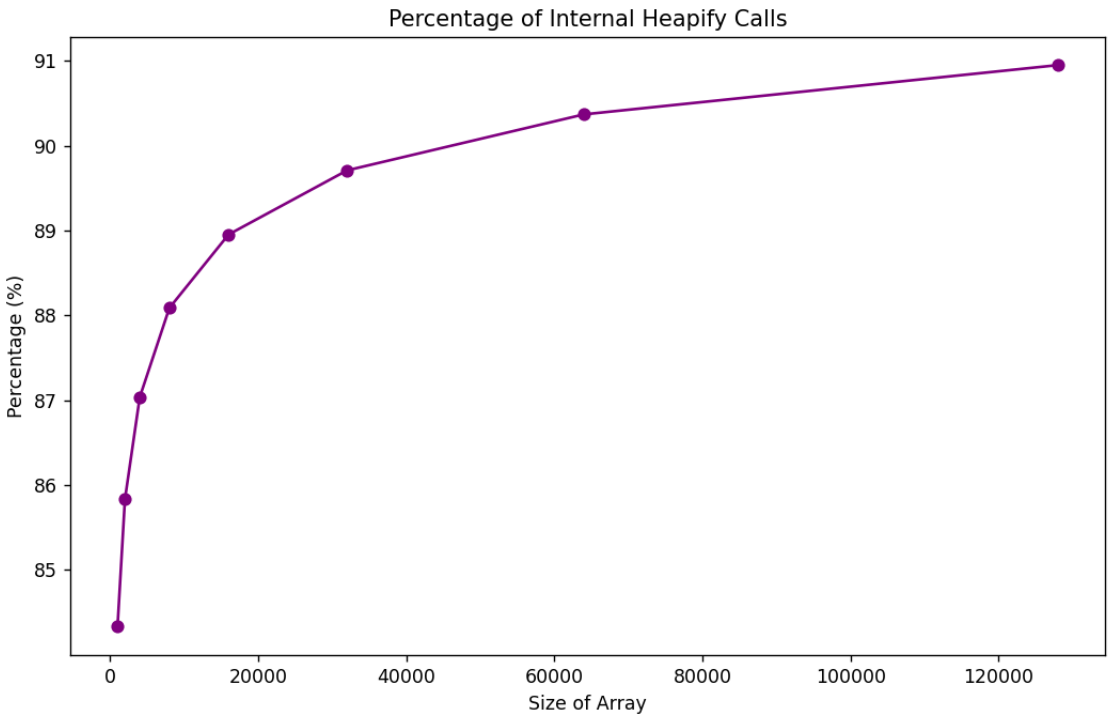
4. График среднего количества вызовов функции heapify и внутренней функции heapify



Зеленая линия (Avg Heapify Calls): Среднее количество вызовов функции `heapify`.
Синяя линия (Avg Internal Heapify Calls): Среднее количество внутренних вызовов функции `heapify`.

Этот график показывает, как часто функция heapify (и ее внутренние вызовы) вызывается в зависимости от размера массива. Это важно для оценки эффективности алгоритма с точки зрения количества операций.

5. График процентного соотношения внутренних вызовов функции heapify к общим вызовам



Фиолетовая линия (Heapify Internal Calls Percentage): Процентное соотношение внутренних вызовов `heapify` от общего числа вызовов.

Этот график помогает понять, насколько часто функция heapify выполняет перестановку элементов в процессе сортировки. В идеале процент этих внутренних вызовов должен быть относительно невысоким, что бы указывать на эффективность алгоритма.

ВЫВОД

В ходе работы мы реализуем пирамидальную сортировку (heap sort) и проводим серию тестов для анализа производительности алгоритма. В ходе тестирования собираются важные метрики, такие как время выполнения, количество вызовов функции heapify, глубина рекурсии и другие параметры. Эти данные затем используются для построения графиков, которые визуализируют поведение алгоритма в зависимости от размера массива.

Код и графики подтверждают, что пирамидальная сортировка работает согласно своей теоретической сложности O(N log N) и эффективно масштабируется при увеличении размеров массива. Графики показывают важные параметры работы алгоритма, такие как время выполнения, глубина рекурсии и количество вызовов функции `heapify`, что позволяет глубже понять его поведение и эффективность в разных ситуациях.