## **Доклад: Алгоритм сортировки Heapsort**

**по предмету Комбинаторика и теория графов**

**Выполнил обучающийся НИТУ МИСИС  
Группа БИВТ-23-1  
Нестеров Никита Михайлович  
Ссылка на реализацию:** [GitHub](https://github.com/akayooo/KITG)

### 

### 

[**Доклад: Алгоритм сортировки Heapsort**](#_xwut6ek7y01m) **1**

[1. Формальная постановка задачи](#_oatyvyeco4wr) 3

[2. Теоретическое описание алгоритма и его характеристики](#_658l4e4goyaw) 4

[Теоретическое описание](#_w5hsbneqzg7z) 4

[Характеристики алгоритма](#_qfhvzhxl17d) 5

[3. Сравнительный анализ Heapsort с аналогичными алгоритмами сортировки](#_dg6zoa7cx2vl) 6

[Heapsort vs. Quicksort](#_twx8p64o5lfc) 6

[Heapsort vs. Mergesort](#_9e59l1vfw3un) 7

[Heapsort vs. Insertion Sort](#_qz8e3v6ismby) 7

[Heapsort vs. Selection Sort](#_7v2uv2weppqi) 8

[4. Перечень инструментов для реализации](#_7077tk2zqjf5) 8

[Языки программирования](#_5e9m0j7o95q1) 8

[Библиотеки и фреймворки](#_qebptqsl06j8) 9

[Инструменты для отладки и тестирования](#_r1w6t965mi41) 9

[Инструменты для визуализации](#_8k1xgrfgdv0a) 10

[Инструменты для анализа производительности](#_e0w5ij7bnffs) 10

[5. Описание реализации и процесса тестирования Heapsort на Python](#_atu6rtt0m17f) 11

[Реализация Heapsort на Python](#_v9boop4y5vmm) 11

[Процесс тестирования](#_cuoky790jy7g) 12

[6. Анализ временной сложности реализованных операций](#_q2koj0oj6yif) 14

[Теоретическая временная сложность](#_tbn9kkyd1hob) 14

[Практические результаты](#_u4qa10haojbj) 14

[Заключение](#_9w6yr6u5942h) 14

### 

### 

### 

### 

### 

### 

### 

### 

### **1. Формальная постановка задачи**

**Задача:**Дан массив (список) чисел A=[a1,a2,…,an]A=[a1​,a2​,…,an​]. Требуется отсортировать элементы массива в порядке неубывания (возрастания), т.е. получить массив A′=[a1′,a2′,…,an′]A′=[a1′​,a2′​,…,an′​] такой, что a1′≤a2′≤⋯≤an′a1′​≤a2′​≤⋯≤an′​.

**Формально:**Дано:

* A=[a1,a2,…,an]A=[a1​,a2​,…,an​] - исходный массив чисел.

Требуется:

* Найти перестановку индексов {1,2,…,n}{1,2,…,n} такую, что aπ(1)≤aπ(2)≤⋯≤aπ(n)aπ(1)​≤aπ(2)​≤⋯≤aπ(n)​.

**Результат:**

* Отсортированный массив A′=[aπ(1),aπ(2),…,aπ(n)]A′=[aπ(1)​,aπ(2)​,…,aπ(n)​].

**Алгоритм Heapsort:**Heapsort решает эту задачу, используя структуру данных "куча" (heap). Куча представляет собой бинарное дерево, в котором каждый родительский узел не меньше (или не больше) своих дочерних узлов. Heapsort использует максимальную кучу (max-heap) для сортировки в порядке возрастания.

**Основные шаги алгоритма:**

1. **Построение кучи:** Преобразуем исходный массив в максимальную кучу.
2. **Сортировка:** Последовательно извлекаем максимальный элемент из кучи и помещаем его в конец массива, уменьшая размер кучи на 1. Повторяем этот процесс до тех пор, пока куча не станет пустой.

### **2. Теоретическое описание алгоритма и его характеристики**

#### **Теоретическое описание**

**Heapsort** — это алгоритм сортировки, основанный на структуре данных "куча" (heap). Куча — это бинарное дерево, удовлетворяющее свойству кучи: для максимальной кучи (max-heap) каждый родительский узел не меньше своих дочерних узлов, а для минимальной кучи (min-heap) — не больше.

**Основные шаги алгоритма Heapsort:**

1. **Построение кучи (Heapify):**
   * Преобразуем исходный массив в максимальную кучу. Это достигается путем просеивания (sift-down) элементов, начиная с последнего родительского узла и двигаясь к корню дерева.
2. **Сортировка:**
   * Последовательно извлекаем максимальный элемент из корня кучи (который является наибольшим элементом) и помещаем его в конец массива.
   * Уменьшаем размер кучи на 1 и восстанавливаем свойство кучи для оставшейся части массива.
   * Повторяем этот процесс до тех пор, пока куча не станет пустой.

#### **Характеристики алгоритма**

**Временная сложность:**

* **Построение кучи (Heapify):**
  + Временная сложность построения кучи составляет O(n), где n — количество элементов в массиве. Это достигается за счет просеивания каждого узла, начиная с последнего родительского узла.
* **Сортировка:**
  + Каждое извлечение максимального элемента и восстановление кучи занимает O(logn) времени.
  + Так как мы выполняем эту операцию nn раз, общая временная сложность сортировки составляет O(nlogn).

Таким образом, **общая временная сложность Heapsort** составляет O(nlogn).

**Пространственная сложность:**

* **Heapsort** — это алгоритм сортировки на месте (in-place), что означает, что он не требует дополнительной памяти, пропорциональной размеру входных данных.
* Единственная дополнительная память, которая используется, — это стек вызовов функций для рекурсии или итеративного восстановления кучи, что составляет O(logn) в худшем случае.

Таким образом, **пространственная сложность Heapsort** составляет O(1) для итеративного подхода и O(logn) для рекурсивного подхода.

**Преимущества и недостатки:**

* **Преимущества:**
  + **Эффективность:** Heapsort имеет временную сложность O(nlogn) в худшем, среднем и лучшем случаях, что делает его очень эффективным для больших наборов данных.
  + **Устойчивость:** Heapsort не требует дополнительной памяти, кроме стека вызовов функций, что делает его устойчивым к ограничениям памяти.
* **Недостатки:**
  + **Неустойчивость:** Heapsort не является устойчивой сортировкой, что означает, что порядок равных элементов может измениться.
  + **Сложность реализации:** Хотя Heapsort эффективен, его реализация может быть более сложной по сравнению с другими алгоритмами сортировки, такими как Quicksort или Mergesort.

### **3. Сравнительный анализ Heapsort с аналогичными алгоритмами сортировки**

#### **Heapsort vs. Quicksort**

**Quicksort:**

* **Временная сложность:** В среднем O(nlogn), но в худшем случае O(n^2) (например, при неудачном выборе опорного элемента).
* **Пространственная сложность:** O(logn) для стека вызовов функций.
* **Устойчивость:** Неустойчивый.
* **Преимущества:** Обычно быстрее на практике, особенно с оптимизациями (например, выбором медианы из трех элементов в качестве опорного).
* **Недостатки:** Худший случай может быть O(n^2), что может быть проблемой для критически важных приложений.

**Heapsort:**

* **Временная сложность:** Всегда O(nlogn).
* **Пространственная сложность:** O(1) для итеративного подхода.
* **Устойчивость:** Неустойчивый.
* **Преимущества:** Гарантированная временная сложность O(nlogn), что делает его надежным выбором для критически важных приложений.
* **Недостатки:** Обычно медленнее Quicksort на практике из-за большего количества сравнений и обменов.

#### **Heapsort vs. Mergesort**

**Mergesort:**

* **Временная сложность:** Всегда O(nlogn).
* **Пространственная сложность:** O(n) для дополнительного массива при слиянии.
* **Устойчивость:** Устойчивый.
* **Преимущества:** Устойчивый, хорошо работает с связными списками.
* **Недостатки:** Требует дополнительной памяти O(n), что может быть проблемой для больших наборов данных.

**Heapsort:**

* **Временная сложность:** Всегда O(nlogn).
* **Пространственная сложность:** O(1) для итеративного подхода.
* **Устойчивость:** Неустойчивый.
* **Преимущества:** Не требует дополнительной памяти, что делает его хорошим выбором для систем с ограниченной памятью.
* **Недостатки:** Неустойчивый, может быть медленнее Mergesort на практике.

#### **Heapsort vs. Insertion Sort**

**Insertion Sort:**

* **Временная сложность:** В среднем и худшем случае O(n^2), в лучшем случае O(n) (когда массив уже отсортирован).
* **Пространственная сложность:** O(1).
* **Устойчивость:** Устойчивый.
* **Преимущества:** Простой в реализации, эффективен для малых наборов данных или почти отсортированных данных.
* **Недостатки:** Неэффективен для больших наборов данных.

**Heapsort:**

* **Временная сложность:** Всегда O(nlogn).
* **Пространственная сложность:** O(1) для итеративного подхода.
* **Устойчивость:** Неустойчивый.
* **Преимущества:** Эффективен для больших наборов данных, не требует дополнительной памяти.
* **Недостатки:** Сложнее в реализации, неустойчивый.

#### **Heapsort vs. Selection Sort**

**Selection Sort:**

* **Временная сложность:** Всегда O(n^2)
* **Пространственная сложность:** O(1).
* **Устойчивость:** Неустойчивый.
* **Преимущества:** Простой в реализации, не требует дополнительной памяти.
* **Недостатки:** Неэффективен для больших наборов данных.

**Heapsort:**

* **Временная сложность:** Всегда O(nlogn).
* **Пространственная сложность:** O(1) для итеративного подхода.
* **Устойчивость:** Неустойчивый.
* **Преимущества:** Эффективен для больших наборов данных, не требует дополнительной памяти.
* **Недостатки:** Сложнее в реализации, неустойчивый.

### **4. Перечень инструментов для реализации**

#### **Языки программирования**

* **Python:**
  + **Преимущества:** Простой синтаксис, богатые библиотеки для работы с массивами и деревьями.
  + **Пример:** Можно использовать встроенные структуры данных и функции для реализации кучи.
* **Java:**
  + **Преимущества:** Объектно-ориентированный подход, широко используется в промышленности.
  + **Пример:** Можно использовать классы и методы для работы с массивами и кучей.
* **C/C++:**
  + **Преимущества:** Близость к аппаратной части, высокая производительность.
  + **Пример:** Можно реализовать кучу с использованием указателей и динамического выделения памяти.
* **JavaScript:**
  + **Преимущества:** Подходит для веб-разработки, интерактивные визуализации.
  + **Пример:** Можно использовать массивы и функции для реализации кучи.
* **Ruby:**
  + **Преимущества:** Чистый и выразительный синтаксис, удобный для быстрой разработки.
  + **Пример:** Можно использовать массивы и методы для реализации кучи.

#### **Библиотеки и фреймворки**

* **Python:**
  + **heapq:** Встроенная библиотека для работы с кучей (min-heap).
  + **numpy:** Библиотека для работы с массивами, может быть полезна для быстрой обработки данных.
* **Java:**
  + **java.util.PriorityQueue:** Класс, реализующий кучу (min-heap).
  + **java.util.Arrays:** Класс с методами для работы с массивами.
* **C++:**
  + **:** Стандартная библиотека с функциями для работы с массивами и кучей.
  + **:** Класс для работы с динамическими массивами.
* **JavaScript:**
  + **Array:** Встроенный объект для работы с массивами.
  + **D3.js:** Библиотека для визуализации данных, может быть полезна для визуализации процесса сортировки.

#### **Инструменты для отладки и тестирования**

* **Отладчики:**
  + **GDB (GNU Debugger):** Для C/C++.
  + **Visual Studio Debugger:** Для C#, C++, Java и других языков.
  + **PyCharm Debugger:** Для Python.
  + **Chrome DevTools:** Для JavaScript.
* **Тестовые фреймворки:**
  + **JUnit:** Для Java.
  + **pytest:** Для Python.
  + **Google Test:** Для C++.
  + **Mocha/Chai:** Для JavaScript.

#### **Инструменты для визуализации**

* **Python:**
  + **Matplotlib:** Библиотека для построения графиков и визуализации данных.
  + **Plotly:** Интерактивная библиотека для визуализации данных.
* **JavaScript:**
  + **D3.js:** Библиотека для визуализации данных, подходит для создания интерактивных визуализаций.
  + **Chart.js:** Простая библиотека для создания графиков.
* **Java:**
  + **JavaFX:** Фреймворк для создания графического интерфейса и визуализации данных.
  + **JFreeChart:** Библиотека для создания графиков и диаграмм.

#### **Инструменты для анализа производительности**

* **Профилировщики:**
  + **cProfile:** Для Python.
  + **VisualVM:** Для Java.
  + **Valgrind:** Для C/C++.
  + **Chrome DevTools Performance:** Для JavaScript.
* **Анализаторы кода:**
  + **SonarQube:** Инструмент для анализа качества кода.
  + **Pylint:** Для Python.
  + **PMD:** Для Java.

### **5. Описание реализации и процесса тестирования Heapsort на Python**

#### **Реализация Heapsort на Python**

**Шаг 1: Построение кучи (Heapify)**

Для построения кучи нам нужна функция heapify, которая будет просеивать элементы вниз, чтобы поддерживать свойство кучи.

def heapify(arr, n, i):

largest = i # Инициализируем наибольший элемент как корень

left = 2 \* i + 1 # Левый потомок

right = 2 \* i + 2 # Правый потомок

# Если левый потомок больше корня

if left < n and arr[left] > arr[largest]:

largest = left

Если правый потомок больше, чем самый большой элемент на данный момент

if right < n and arr[right] > arr[largest]:

largest = right

# Если самый большой элемент не корень

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i] # Меняем местами

heapify(arr, n, largest) # Рекурсивно просеиваем вниз

**Шаг 2: Основная функция Heapsort**

Теперь мы можем реализовать основную функцию heapsort, которая будет использовать heapify для сортировки массива.

def heapsort(arr):

n = len(arr)

Построение максимальной кучи

for i in range(n // 2 - 1, -1, -1):

heapify(arr, n, i)

Извлечение элементов из кучи по одному

for i in range(n - 1, 0, -1):

arr[i], arr[0] = arr[0], arr[i] # Меняем местами

heapify(arr, i, 0) # Просеиваем вниз

**Пример использования**

arr = [12, 11, 13, 5, 6, 7]

heapsort(arr)

print("Отсортированный массив:", arr)

#### **Процесс тестирования**

**1. Ручное тестирование**

Можно вручную проверить работу алгоритма на нескольких примерах:

arr1 = [3, 6, 8, 10, 1, 2]

heapsort(arr1)

print("Отсортированный массив 1:", arr1)

arr2 = [5, 3, 8, 4, 2, 7, 1, 10]

heapsort(arr2)

print("Отсортированный массив 2:", arr2)

arr3 = [1]

heapsort(arr3)

print("Отсортированный массив 3:", arr3)

**2. Автоматизированные тесты с использованием pytest**

Для более систематического тестирования можно использовать фреймворк pytest:

import pytest

def test\_heapsort():

assert heapsort([3, 6, 8, 10, 1, 2]) == [1, 2, 3, 6, 8, 10]

assert heapsort([5, 3, 8, 4, 2, 7, 1, 10]) == [1, 2, 3, 4, 5, 7, 8, 10]

assert heapsort([1]) == [1]

assert heapsort([]) == []

assert heapsort([1, 1, 1, 1]) == [1, 1, 1, 1]

assert heapsort([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

if \_\_name\_\_ == "\_\_main\_\_":

pytest.main()

**3. Профилирование производительности**

Для анализа производительности можно использовать модуль cProfile:

import cProfile

arr = [12, 11, 13, 5, 6, 7]

cProfile.run('heapsort(arr)')

Это позволит увидеть, сколько времени затрачивается на каждую функцию и поможет выявить узкие места в реализации.

### **6. Анализ временной сложности реализованных операций**

#### **Теоретическая временная сложность**

* **Построение кучи (Heapify):** O(n)
* **Сортировка:** O(nlogn)

#### **Практические результаты**

Для анализа практической производительности можно использовать профилирование с помощью cProfile. Пример:

import cProfile

arr = [12, 11, 13, 5, 6, 7]

cProfile.run('heapsort(arr)')

Результаты профилирования покажут, сколько времени затрачивается на каждую функцию, что позволит сравнить практические результаты с теоретическими оценками.

### **Заключение**

Heapsort — это алгоритм сортировки с гарантированной временной сложностью O(nlogn) и пространственной сложностью O(1), что делает его надежным выбором для больших наборов данных и систем с ограниченной памятью. Однако, он не является устойчивым и может быть медленнее некоторых других алгоритмов, таких как Quicksort, на практике. Выбор алгоритма сортировки зависит от конкретных требований задачи, таких как размер данных, требования к устойчивости, доступная память и другие факторы.