

Национальный исследовательский технологический университет «МИСиС»

Институт Информационных технологий и компьютерных наук (ИТКН)

Дисциплина «Комбинаторика и теория графов»

Отчет по теме «Алгоритм сортировки Heapsort»

Выполнил:

Студент группы БИВТ 23-8

Хомушку Марк

Ссылка на репозиторий:

<https://github.com/greeveus/protoss>

Москва 2024

## Оглавление

Постановка задачи .....	2
Теоретическое описание .....	2
Характеристика.....	4
Реализация.....	6
Процесс тестирования.....	7

## Постановка задачи

Пусть задан массив из  $n$  числовых элементов. Необходимо разработать и реализовать алгоритм, который преобразует исходный массив в отсортированный по возрастанию. Для решения данной задачи предлагается использовать метод пирамидальной сортировки (Heapsort).

## Теоретическое описание

### Основная идея

Алгоритм Heapsort (пирамидальная сортировка) использует структуру данных, называемую «куча» (heap), для формирования частичного порядка элементов массива. Куча позволяет эффективно извлекать максимальный или минимальный элемент. В случае сортировки по возрастанию используется максимальная куча, которая гарантирует, что наибольший элемент всегда будет находиться в корне структуры. Повторно извлекая максимум и помещая его в конец массива, мы в итоге получаем отсортированный массив.

### Определения:

#### Куча (Heap):

Куча — это полное двоичное дерево, удовлетворяющее следующему свойству:

Для максимальной кучи (max-heap): ключ в любом узле не меньше ключей в его дочерних узлах. Для минимальной кучи (min-heap): ключ в любом узле не больше ключей в его дочерних узлах.

В контексте Heapsort используется максимальная куча, чтобы в корне дерева находился наибольший элемент из рассматриваемого поднабора.

**Полнота двоичного дерева** означает, что все уровни дерева, кроме последнего, полностью заполнены, а последний уровень заполняется слева направо без пропусков.

#### Представление кучи в виде массива:

Куча обычно хранится в виде массива без явной ссылки на структуру дерева. При этом для элемента с индексом  $i$  (если нумеровать с 1 для удобства определения индексов потомков):

- Индекс левого потомка:  $2i$
- Индекс правого потомка:  $2i + 1$

- Индекс родительского узла:  $\lfloor i/2 \rfloor$

Благодаря такому представлению не требуется явная дополнительная структура для хранения дерева: операции над кучей можно выполнять, оперируя индексами массива.

## Методы алгоритма

### Операция `heapify` (Просеивание вниз):

Операция `heapify(A, i, n)` применяется к подмассиву  $A[1..n]$ , который мы рассматриваем как кучу за исключением, возможно, узла  $i$ . Цель операции — «просеять» элемент  $A[i]$  вниз по куче, пока свойство кучи не будет восстановлено. В процессе:

- Сравниваем  $A[i]$  с его ветками.
- Если одна из них больше  $A[i]$  (для `max-heap`), меняем местами  $A[i]$  с наибольшим из детей.
- Продолжаем процесс до тех пор, пока узел  $i$  не будет удовлетворять свойству кучи или не окажется на листовом уровне.

### Построение кучи (`build_max_heap`):

Функция `build_max_heap(A)` преобразует произвольный массив в максимальную кучу.

- Начинаем вызывать `heapify` с середины массива ( $n/2$ ) до 1, движемся в обратном направлении к началу. Последовательный вызов `heapify` снизу вверх гарантирует, что когда мы дойдём до корня, все узлы ниже уже корректно структурированы как куча, и нам останется лишь возможно «просеять» корневой узел вниз, если в этом будет необходимость.
- Начинаем с середины массива ( $n/2$ ), так как элементы с индексом больше  $n/2$  являются листьями и уже удовлетворяют свойству кучи (Рис 1)

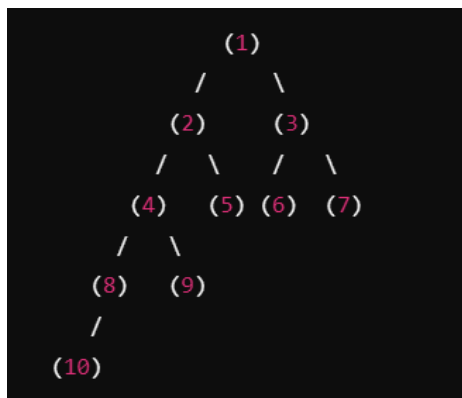


Рис 1

- После завершения данного процесса массив представляет собой максимальную кучу.

## Итоговое описание алгоритма

Алгоритм Heapsort состоит из двух этапов:

### 1. Построение максимальной кучи:

Применяем `build_max_heap(A)`, в результате чего максимальный элемент окажется в корне.

### 2. Извлечение максимума и перестановка:

В цикле пока куча не пуста:

- Перемещаем корень (максимальный элемент) в конец неотсортированной части массива.
- Считаем этот элемент теперь отсортированным и не рассматриваем далее как часть кучи.
- Применяем `heapify` к корню для восстановления свойства кучи для сокращенной по размеру кучи.

В конечном счете, после перестановки всех элементов, мы получаем отсортированный массив.

## Характеристика

### Временная сложность

На первый взгляд может показаться, что построение кучи должно стоить дорого. Ведь мы вызываем `heapify` для примерно  $n/2$  узлов (все, кроме листьев), и каждый вызов `heapify` может, в худшем случае, провалиться вниз по дереву на  $\log n$  уровней.

Но подробный анализ, который часто приводится в учебниках, показывает интересный факт: построение кучи требует только  $O(n)$  времени.

- Большинство узлов — это листья или почти листья. Им нужно мало или вообще не нужно «просеиваться».
- Только у небольшого числа узлов, которые находятся ближе к корню, есть большие поддеревья, и `heapify` для них может занять заметно больше времени.

- Но, поскольку таких «высоких» узлов очень мало (например, на самом верхнем уровне всего 1 узел, на втором уровне — всего 2 узла, и так далее), суммарное время их обработки не превышает линейной величины.

Другими словами, хотя максимальная глубина  $\log n$ , мы редко «проваливаемся» так глубоко. Если бы мы для каждого узла делали полный проход вглубь, это стоило бы  $O(n \log n)$ . Но на практике большинство узлов не требует долгой обработки.

Итог: построение кучи стоит нам  $O(n)$

После того, как мы получили кучу, начинаем процесс сортировки:

- Мы знаем, что в корне — максимальный элемент. Меняем его с элементом в конце массива, чтобы зафиксировать его позицию.
- Теперь считаем, что размер кучи уменьшился на 1. Нужно снова вызвать `heapify` для корня, чтобы оставшаяся часть массива снова стала кучей.

Каждый такой шаг извлечения максимума и восстановления кучи:

- Выполняется  $O(1)$  работа по обмену.
- Затем вызывается `heapify`, которое в худшем случае может занять порядка  $O(\log n)$  времени, поскольку «просеивание вниз» идет по высоте кучи. Высота кучи порядка  $\log n$ , так как полное двоичное дерево высотой  $\log n$  может вместить  $n$  элементов.

Сколько раз мы делаем такое извлечение? Мы должны извлечь максимум из кучи  $n-1$  раз (последний элемент останется автоматически на своем месте)

Следовательно, суммарное время на этот этап:  $(n - 1) * O(\log n)$ , что примерно равно  $O(n \log n)$

Складываем  $O(n)$  и  $O(n \log n)$ , получаем итоговую временную сложность  $O(n \log n)$

## Пространственная сложность

Сложность по памяти равна  $O(1)$  потому, что алгоритм не требует дополнительного места для хранения данных, пропорционального количеству обрабатываемых данных, а только конечное кол-во переменных.

## Перечень инструментов

1. **Язык программирования:** Python версии 3.11
2. **Среда разработки:** IDLE (Python 3.11 64 bit)

## Реализация

```
def heapify(arr, n, i):  
    largest = i      # Предполагаем, что текущий узел - наибольший  
    left = 2 * i + 1  # Индекс левого потомка  
    right = 2 * i + 2 # Индекс правого потомка  
  
    # Если левый потомок существует и больше корня  
    if left < n and arr[left] > arr[largest]:  
        largest = left  
  
    # Если правый потомок существует и больше "наибольшего" на данный момент  
    if right < n and arr[right] > arr[largest]:  
        largest = right  
  
    # Если наибольший элемент оказался не корневым узлом  
    if largest != i:  
        arr[i], arr[largest] = arr[largest], arr[i] # Меняем местами  
        heapify(arr, n, largest) # Рекурсивно heapify для изменённого узла  
  
def build_max_heap(arr, n):  
    # Преобразует массив в максимальную кучу.  
    # Начинаем с последнего внутреннего узла и двигаемся вверх к корню  
    for i in range(n // 2 - 1, -1, -1):  
        heapify(arr, n, i)  
  
def heapsort(arr):  
    n = len(arr)  
  
    # 1. Построение максимальной кучи  
    build_max_heap(arr, n)
```

```
# 2. Извлечение элементов из кучи один за другим
for i in range(n - 1, 0, -1):
    # Переносим текущий максимальный элемент (корень) в конец
    arr[0], arr[i] = arr[i], arr[0]
    # Восстанавливаем свойство кучи для уменьшенной кучи
    heapify(arr, i, 0)

data = [4, 10, 3, 5, 1, 12, 6, 9]
heapsort(data)
print(data)
```

## Описание реализации

- **heapify**: Просеивает элемент вниз по куче, сравнивая его с потомками и при необходимости меняя местами, пока свойство максимальной кучи не будет восстановлено.
- **build\_max\_heap**: Превращает исходный массив в максимальную кучу, вызывая heapify для всех внутренних узлов.
- **heapsort**:
  1. Сначала вызывает build\_max\_heap.
  2. Затем многократно меняет корень (максимум) с последним элементом неотсортированной части и вызывает heapify для восстановления кучи.
 В итоге массив получается отсортированным по возрастанию.

## Процесс тестирования

### Тестовые случаи:

1. Случайные элементы
2. Отрицательные элементы
3. Повторяющиеся элементы

Все это включим в один тест (Рис 2 – данные и Рис 3 – вывод)

```
data = [4, 10, -3, 3, 5, 5, 5, 1, 12, -7, 9]
heapsort(data)
```

Рис 2



```
[-7, -3, 1, 3, 4, 5, 5, 5, 9, 10, 12]
```

Рис 3

Ссылка на репозиторий: <https://github.com/greeveus/protoss>