

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №
по курсу «Алгоритмы и структуры данных»
Тема: Деревья. Пирамида, пирамидальная сортировка.
Очередь с приоритетами.

Вариант 21

Выполнила:
Савченко А.С.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №1. Куча ли?	3
Задача №4. Построение пирамиды	5
Задача №7. Снова сортировка	9
Вывод (по всей лабораторной)	14

Задачи по варианту

Задача №1. Куча ли?

Текст задачи:

1 задача. Куча ли?

Структуру данных «куча», или, более конкретно, «неубывающая пирамида», можно реализовать на основе массива.

Для этого должно выполняться основное свойство неубывающей пирамиды, которое заключается в том, что для каждого $1 \leq i \leq n$ выполняются условия:

1. если $2i \leq n$, то $a_i \leq a_{2i}$,
2. если $2i + 1 \leq n$, то $a_i \leq a_{2i+1}$.

Дан массив целых чисел. Определите, является ли он неубывающей пирамидой.

- **Формат входного файла (input.txt).** Первая строка входного файла содержит целое число n ($1 \leq n \leq 10^6$). Вторая строка содержит n целых чисел, по модулю не превосходящих $2 \cdot 10^9$.
- **Формат выходного файла (output.txt).** Выведите «YES», если массив является неубывающей пирамидой, и «NO» в противном случае.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Примеры:

№	input.txt	output.txt
1	5 1 0 1 2 0	NO
2	5 1 3 2 5 4	YES

Листинг кода:

```
def valid_heap(arr):  
    n = len(arr)  
    for i in range(n // 2):  
        if 2 * i + 1 < n and arr[i] > arr[2 * i + 1]:  
            return "NO"  
        if 2 * i + 2 < n and arr[i] > arr[2 * i + 2]:  
            return "NO"
```

```

return "YES"

if __name__ == "__main__":
    with open('input.txt', 'r') as input_file:
        n = int(input_file.readline().strip())
        arr = list(map(int,
            input_file.readline().strip().split()))

    ans = valid_heap(arr)

    with open('output.txt', 'w') as output_file:
        output_file.write(ans)

```

Текстовое объяснение решения:

Идея: Узел i имеет детей, если $2 * i + 1 < n$ и $2 * i + 2 < n$

--> чтобы проверить все узлы, имеющие детей, нужно пройти только до половины массива, последующий либо являются листьями, либо у них нет детей, поэтому проверять дальше нет смысла.

Сам алгоритм: Проходим в цикле по элементам массива до середины, и проверяем условие пирамиды для левого ребенка (если его индекс $2 * i + 1$ меньше n длины массива (убедились, что существует левый ребенок) и проверяем свойства неубывающей пирамиды если $arr[i] > arr[2 * i + 1]$ возвращаем "NO" тк нарушается усл неуб пирамиды (родительский узел должно быть \leq его левого ребенка). Аналогично для правого ребенка.

Результат работы кода на примерах из текста задачи:(скрины input output файлов):

input.txt			:	output.txt		
1	5	✓		1	NO	
2	1 0 1 2 0					

input.txt			:	output.txt		
1	5	✓		1	YES	
2	1 3 2 5 4					

Результат работы кода на минимальных и максимальных значениях:(скрины input output файлов):

input.txt			output.txt	
1	1	✓	1	YES
2	1			

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0008549 с	14.76171875 Мб
Пример из задачи	0.0008097 с	14.671875 Мб
Верхняя граница диапазона значений входных данных из текста задачи	0.302281 с	17.93359375 Мб

Вывод по задаче:

Был реализован алгоритм проверяющий является ли массив неубывающей пирамидой (двоичной кучей) проверяя условие. Алгоритм эффективен его сложность $O(n)$ что позволяет ему укладываться в ограничения времени и памяти.

Задача №4. Построение пирамиды

Текст задачи:

4 задача. Построение пирамиды

В этой задаче вы преобразуете массив целых чисел в пирамиду. Это важнейший шаг алгоритма сортировки под названием HeapSort. Гарантированное время работы в худшем случае составляет $O(n \log n)$, в отличие от *среднего* времени работы QuickSort, равного $O(n \log n)$. QuickSort обычно используется на практике, потому что обычно он быстрее, но HeapSort используется для внешней сортировки, когда вам нужно отсортировать огромные файлы, которые не помещаются в памяти вашего компьютера.

Первым шагом алгоритма HeapSort является создание пирамиды (heap) из массива, который вы хотите отсортировать.

Ваша задача - реализовать этот первый шаг и преобразовать заданный массив целых чисел в пирамиду. Вы сделаете это, применив к массиву определенное количество перестановок (swaps). Перестановка - это операция, как вы помните, при которой элементы a_i и a_j массива меняются местами для некоторых i и j . Вам нужно будет преобразовать массив в пирамиду, используя только $O(n)$ перестановок. Обратите внимание, что в этой задаче вам нужно будет использовать min-heap вместо max-heap.

- **Формат ввода или входного файла (input.txt).** Первая строка содержит целое число n ($1 \leq n \leq 10^5$), вторая содержит n целых чисел a_i входного массива, разделенных пробелом ($0 \leq a_i \leq 10^9$, все a_i - различны.)
- **Формат выходного файла (output.txt).** Первая строка ответа должна содержать целое число m - количество сделанных свопов. Число m должно удовлетворять условию $0 \leq m \leq 4n$. Следующие m строк должны содержать по 2 числа: индексы i и j сделанной перестановки двух элементов, **индексы считаются с 0**. После всех перестановок в нужном порядке массив должен стать пирамидой, то есть для каждого i при $0 \leq i \leq n-1$ должны выполняться условия:

1. если $2i + 1 \leq n - 1$, то $a_i < a_{2i+1}$,
2. если $2i + 2 \leq n - 1$, то $a_i < a_{2i+2}$.

Обратите внимание, что все элементы входного массива различны. Любая последовательность свопов, которая менее $4n$ и после которой входной массив становится корректной пирамидой, считается верной.

- Ограничение по времени. 3 сек.
- Ограничение по памяти. 512 мб.
- Пример 1:

input.txt	output.txt
5	3
5 4 3 2 1	1 4
	0 1
	1 3

После перестановки элементов в позициях 1 и 4 массив становится следующим: 5 1 3 2 4.

Далее, перестановка элементов с индексами 0 и 1: 1 5 3 2 4. И напоследок, переставим 1 и 3: 1 2 3 5 4, и теперь это корректная неубывающая пирамида.

- Пример 2:

input.txt	output.txt
5	0
1 2 3 4 5	

Листинг кода:

```
def min_heapify(A, n, i, swaps):
    left = 2 * i + 1
    right = 2 * i + 2
    smallest = i
    if left < n and A[left] < A[smallest]:
        smallest = left

    if right < n and A[right] < A[smallest]:
        smallest = right

    if smallest != i:
        A[i], A[smallest] = A[smallest], A[i]
        swaps.append((i, smallest))
        min_heapify(A, n, smallest, swaps)

def build_min_heap(A):
    n = len(A)
    swaps = []
    for i in range(n // 2 - 1, -1, -1):
        min_heapify(A, n, i, swaps)
    return swaps

if __name__ == "__main__":
    with open('input.txt', 'r') as input_file:
        n = int(input_file.readline().strip())
        arr = list(map(int,
input_file.readline().strip().split()))

    swaps = build_min_heap(arr)

    with open('output.txt', 'w') as output_file:
        output_file.write(f"{len(swaps)}\n")
```

```
output_file.writelines(f"{i} {j}\n" for i, j in swaps)
```

Текстовое объяснение решения:

Чуть перепишем код из лекции чтобы было min-heap вместо max-heap. Напишем функцию `min_heapify` которая восстанавливает свойство min-heap: предполагаем что `i` - минимальный узел, если это не так, проверяем существует ли левый ребенок и если он меньше обновляем значение `i` на `left`, тоже для правого ребенка.

Когда свойство min-heap восстановлено, выполняем перестановки, если узел не наименьшей, при этом сохраняем перестановки кортежами в список `swaps`, рекурсивно вызываем `min_heapify` для поддерева.

Результат работы кода на примерах из текста задачи:(скрины input output файлов):

input.txt	output.txt
1 5	1 3
2 5 4 3 2 1	2 1 4
	3 0 1
	4 1 3

input.txt	output.txt
1 5	1 0
2 1 2 3 4 5	2

Результат работы кода на минимальных и максимальных значениях:(скрины input output файлов):

input.txt	output.txt
1 100000	74476 1 5
2 419604904	74477 3 8
	74478 8 18
	74479 18 37
	74480 37 76
	74481 76 154
	74482 154 310
	74483 310 621
	74484 621 1243
	74485 1243 2487
	74486 2487 4976
	74487 4976 9954
	74488 9954 19910
	74489 19910 39821
	74490 39821 79644
	74491

input.txt	output.txt
1 1	1 0
2 1	2

	Время выполнения	Затраты памяти
--	------------------	----------------

Нижняя граница диапазона значений входных данных из текста задачи	0.0007914 с	14.97265625 Мб
Пример из задачи	0.0009864 с	15.0625 Мб
Верхняя граница диапазона значений входных данных из текста задачи	0.1241312 с	20.8828125 Мб

Вывод по задаче:

Для реализации алгоритма были преобразованы функции из лекции. Алгоритм показывает какие перестановки надо выполнить чтобы массив стал min heap. Оценка времени $O(n)$. Алгоритм работает эффективно и отлично справляется с максимальными входными данными из условия.

Задача №7. Снова сортировка

Текст задачи:

7 задача. Снова сортировка

Напишите программу пирамидальной сортировки на Python для последовательности в **убывающем порядке**. Проверьте ее, создав несколько случайных массивов, подходящих под параметры:

- **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 10^5$) — число элементов в массиве. Во второй строке находятся n различных целых чисел, *по модулю* не превосходящих 10^9 .

- **Формат выходного файла (output.txt).** Одна строка выходного файла с отсортированным по невозрастанию массивом. Между любыми двумя числами должен стоять ровно один пробел.
- Для проверки можно выбрать случай, когда сортируется массив размера $10^3, 10^4, 10^5$ чисел порядка 10^9 , отсортированных в обратном порядке; когда массив уже отсортирован в нужном порядке; когда много одинаковых элементов, всего 4-5 уникальных; средний - случайный. Сравните на данных сетах Randomized-QuickSort, MergeSort, HeapSort, InsertionSort.
- Есть ли случай, когда сортировка пирамидой выполнится за $O(n)$?
- * Напишите процедуру Max-Heapify, в которой вместо рекурсивного вызова использовалась бы итеративная конструкция (цикл).

Листинг кода:

```
def max_heapify(arr, n, i):
    while i < n:
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2

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

        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            i = largest
        else:
            break

def build_max_heap(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)
```

```

def heap_sort(arr):
    n = len(arr)
    build_max_heap(arr)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        max_heapify(arr, i, 0)

if __name__ == "__main__":
    with open('input.txt', 'r') as input_file:
        n = int(input_file.readline().strip())
        arr = list(map(int,
            input_file.readline().strip().split()))

    heap_sort(arr)

    with open('output.txt', 'w') as output_file:
        output_file.write(" ".join(map(str, arr)))

```

Текстовое объяснение решения:

Что ж задача написать пирамидальную сортировку `heap_sort`. Она тоже была в лекции, но тут мы еще напишем процедуру `max_heapify` итеративно.

В функции `max_heapify` восстанавливаем свойство `max-heap`; `build_max_heap`: преобразуем массив в кучу `max-heap`. Наконец в `build_max_heap` проходим по всем внутренним узлам, начиная с середины массива ($n // 2 - 1$) до начала (т.е. Шаг -1), и вызываем функцию `max_heapify`. Выводим согласно формату выходного файла.

Результат работы кода на примерах из текста задачи:(скрины input output файлов):

input.txt	
1	10
2	130630398 -498629441 178198648 -247435938 532070198 870859299 -176850677 653146847 -74164898 326854990
3	
output.txt	
1	-498629441 -247435938 -176850677 -74164898 130630398 178198648 326854990 532070198 653146847 870859299
2	

Результат работы кода для разных массивов и разных сортировок:

```
HeapSort: 0.001988 сек
QuickSort: 0.000993 сек
MergeSort: 0.001000 сек
InsertionSort: 0.000999 сек
Тест sorted на массиве размером 1000
HeapSort: 0.001000 сек
QuickSort: 0.002000 сек
MergeSort: 0.001023 сек
InsertionSort: 0.040975 сек
Тест few_unique на массиве размером 1000
HeapSort: 0.000998 сек
QuickSort: 0.009705 сек
MergeSort: 0.001002 сек
InsertionSort: 0.015078 сек
Тест random на массиве размером 1000
HeapSort: 0.001910 сек
QuickSort: 0.001096 сек
MergeSort: 0.000940 сек
InsertionSort: 0.018959 сек
Тест reverse_sorted на массиве размером 10000
HeapSort: 0.025372 сек
QuickSort: 0.015006 сек
MergeSort: 0.014005 сек
InsertionSort: 0.000993 сек
Тест sorted на массиве размером 10000
HeapSort: 0.028904 сек
QuickSort: 0.019000 сек
MergeSort: 0.017102 сек
InsertionSort: 3.843698 сек
Тест few_unique на массиве размером 10000
HeapSort: 0.021025 сек
QuickSort: 0.367486 сек
MergeSort: 0.017015 сек
InsertionSort: 1.579333 сек
Тест random на массиве размером 10000
HeapSort: 0.025638 сек
QuickSort: 0.015000 сек
MergeSort: 0.018000 сек
InsertionSort: 1.979527 сек
Тест reverse_sorted на массиве размером 100000
HeapSort: 0.316773 сек
QuickSort: 0.200984 сек
MergeSort: 0.180902 сек
InsertionSort: 0.009999 сек
Тест sorted на массиве размером 100000
HeapSort: 0.363683 сек
QuickSort: 0.169952 сек
MergeSort: 0.177013 сек
Тест few_unique на массиве размером 100000
HeapSort: 0.276610 сек
QuickSort: 4.331804 сек
MergeSort: 0.213627 сек
InsertionSort: не справляется тк имеет временную сложность  $O(n^2)$  и неэффективен
Тест random на массиве размером 100000
HeapSort: 0.383411 сек
QuickSort: 0.190470 сек
MergeSort: 0.226589 сек
InsertionSort: не справляется тк имеет временную сложность  $O(n^2)$  и неэффективен
```

Вывод по тестам:

MergeSort:

Временная сложность: $O(n \log n)$.

Память: Использует дополнительную память $O(n)$.

Справится ли: MergeSort может справиться с массивами размером 10^9 элементов, однако требует достаточно много дополнительной памяти.

Для чего подходит: Стабильная, идеально для больших массивов

HeapSort:

Временная сложность: $O(n \log n)$.

Память: Использует $O(1)$ дополнительной памяти.

Справится ли: HeapSort справится с массивами любого из данных размеров эффективности.

QuickSort (нужна оптимизация по глубине рекурсии):

Временная сложность: $O(n \log n)$ в среднем.

Память: Использует $O(\log n)$ дополнительной памяти в среднем.

Справится ли: Да, с оптимизацией глубины рекурсии

Для чего подходит: Лучший выбор для случайных массивов. Может показывать плохие результаты на массивах с небольшим числом уникальных элементов и в худшем случае $O(n^2)$.

InsertionSort:

Временная сложность: $O(n^2)$.

Память: Использует $O(1)$ дополнительной памяти.

Справится ли с массивами размером 10^5 и 10^9 : Нет, InsertionSort неэффективен

Вывод по задаче:

Для данной задачи была реализована пирамидальная сортировки массива в убывающем порядке. Алгоритм состоит из построения кучи `build_max_heap`, итеративного `max_heapify`, и самой сортировки `heap_sort`.

Сравнив разные виды сортировок на разных входных данных можно сказать: MergeSort и HeapSort -стабильные и эффективные для больших массивов из-за $O(n \log n)$. QuickSort также хорош, лучший выбор для случайных массивов,но может показывать плохие результаты на массивах с небольшим числом уникальных элементов и в худшем случае $O(n^2)$. InsertionSort подходит только для небольших массивов или частично отсортированных данных. MergeSort подходит, если есть достаточное количество памяти. HeapSort - лучший выбор, если нужно минимальное использование памяти.

Вывод (по всей лабораторной)

Я результате выполнения лабораторной работы я не только познакомилась с кучей и пирамидальной сортировкой, но и провела анализ многих изученных сортировок, а также сделала вывод какая сортировка для каких наборов данных является наиболее эффективной.