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

Отчет по лабораторной работе №2

по курсу «Алгоритмы и структуры данных»

Тема: Быстрая сортировка, сортировки за линейное время

Вариант 15

Выполнил:
Скворцов Д.А.
К3140

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №1. Улучшение Quick sort

Используя псевдокод процедуры Randomized - QuickSort, а так же Partition из презентации к Лекции 3 (страницы 8 и 12), напишите программу быстрой сортировки на Python и проверьте ее, создав несколько рандомных массивов, подходящих под параметры

- **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 10^4$) — 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 массивом. Между любыми двумя числами должен стоять ровно один пробел.
- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб.
- Для проверки можно выбрать наихудший случай, когда сортируется массив размера 10^3 , 10^4 , 10^5 чисел порядка 10^9 , отсортированных в обратном порядке; наилучший, когда массив уже отсортирован, и средний - случайный. Сравните на данных сетах Randomized-QuickSort и простой QuickSort. (А также есть Median-QuickSort, см. задание 10.2; и Tail-Recursive-QuickSort, см. Кормен. 2013, стр. 217)

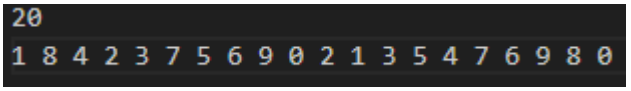
2. Основное задание. Цель задачи - переделать данную реализацию рандомизированного алгоритма быстрой сортировки, чтобы она работала быстро даже с последовательностями, содержащими много одинаковых элементов. Чтобы заставить алгоритм быстрой сортировки эффективно обрабатывать последовательности с несколькими уникальными элементами, нужно заменить двухстороннее разделение на трехстороннее (смотри в Лекции 3 слайд 17). То есть ваша новая процедура разделения должна разбить массив на три части.

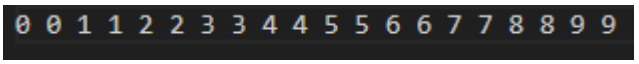
Листинг кода.

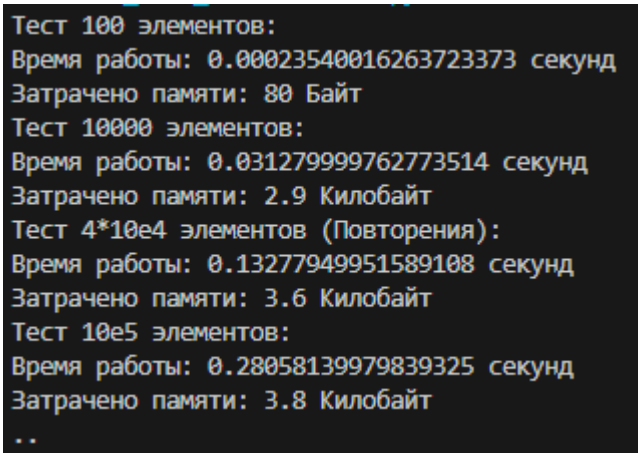
```
def Partition3(A, l, r):
    x = A[l]
    i = l + 1
    m2 = 1
    for j in range(l + 1, r + 1):
```

```
        if A[j] <= x:
            if A[j] == x:
                m2 += 1
            A[i], A[j] = A[j], A[i]
            i += 1

    p = i - 1
    k = l
    c = 0
    while k < p:
        if A[k] != x:
            k += 1
            continue
        while p > k and A[p] == x:
            p -= 1

        A[k] , A[p] = A[p] , A[k]
        k += 1
        c += 1
        if c == m2:
            break
    m1 = p
    m2 += m1 - 1
    return m1, m2

def RandomizedQuickSort3(A, l, r):
    if l < r:
        k = randint(l, r)
        A[l], A[k] = A[k], A[l]
        m1, m2 = Partition3(A, l, r)

        RandomizedQuickSort3(A, l, m1 - 1)
        RandomizedQuickSort3(A, m2 + 1, r)
```

Алгоритм рандомизированной быстрой сортировки реализован двумя функциями: *RandomizedQuickSort3()* и *Partition3()*. Первая отвечает за выбор случайного элемента и запуска ветвления по половинам, когда как вторая перемешивает массив относительно ключевого элемента. Также по заданию добавлена оптимизация для равных элементов - часть с ними не включается в общую сортировку, что немного ускоряет работу.







	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0002354001626372337 3 секунд	80 байт
Медиана диапазона значений входных данных из текста задачи	0.13277949951589108 секунд	3.6 Килобайт
Верхняя граница диапазона значений входных данных из текста задачи	0.28058139979839325 секунд	3.8 Килобайт

Вывод по задаче:

Алгоритм реализован корректно и даже на данных порядка 10^5 выполняется за время, меньшее максимально данного. По сравнению с алгоритмом обычной быстрой сортировки, этот вариант с меньшей вероятностью будет попадать на худший по времени случай. А оптимизация ускоряет программу на однотипных данных.

Задача №2. Анти-quick sort

Для сортировки последовательности чисел широко используется быстрая сортировка - QuickSort. Далее приведена программа на языке Python, которая сортирует массив a, используя этот алгоритм.

Хотя QuickSort является очень быстрой сортировкой в среднем, существуют тесты, на которых она работает очень долго. Оценивать время работы алгоритма будем числом сравнений с элементами массива (то есть, суммарным числом сравнений в первом и втором while). Требуется написать программу, генерирующую тест, на котором быстрая сортировка сделает наибольшее число таких сравнений.

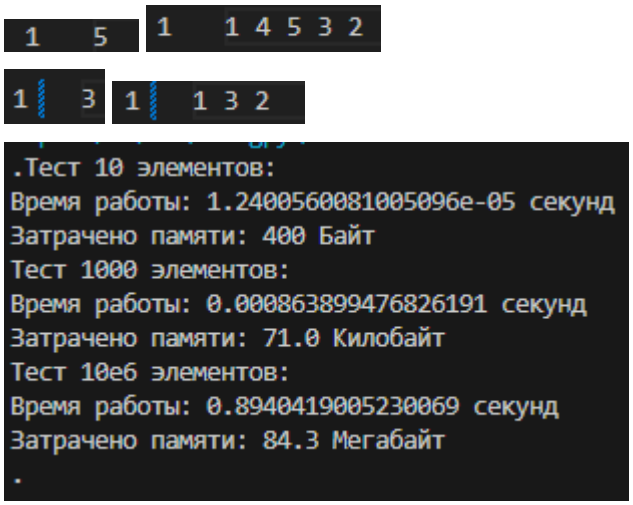
- **Формат входного файла (input.txt).** В первой строке находится единственное число n ($1 \le n \le 10^6$).
- **Формат выходного файла (output.txt).** Вывести перестановку чисел от 1 до n, на которой быстрая сортировка выполнит максимальное число сравнений. Если таких перестановок несколько, вывести любую из них.
- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб.

Листинг кода.

```python
def solution(n): # reverse qsort
    if n <= 1:
        return list(range(1, n+1))
    lst = list(range(n))
    r = n-1
    indices = []

    while 2 <= r:
        mid = r // 2 # l = 0
        indices.append(lst[mid])
        lst[r], lst[mid] = lst[mid], lst[r]
        r -= 1
    ans = [0]*n
    for n, idx in enumerate(lst):
        ans[idx] = n+1
    return ans
```

Функция *solution()* разворачивает процесс сортировки в худший случай - когда *quicksort* выбирает за ключевой элемент - максимальный. Таким

образом каждый разделение массива происходит неэффективно, и финальное время становится сравнимым с n^2.



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0000124005600810 05 секунд	400 байт
Медиана диапазона значений входных данных из текста задачи	0.000863899476826191 секунд	70.9 Килобайт
Верхняя граница диапазона значений входных данных из текста задачи	0.8940419005230069 секунд	84.3 Мегабайт

Вывод по задаче:
Алгоритм работает корректно: время выполнения в самом крайнем случае не выходит за рамки, но довольно велико, поскольку асимптотическая сложность алгоритма = *O(n)*. Каждый раз генерируется последовательность, являющаяся худшим случаем для предложенной сортировки, так как подбирается самый неэффективный случай расстановки.

Задача №4. Точки и отрезки

Цель. Вам дается набор точек и набор отрезков. Цель состоит в том, чтобы вычислить для каждой точки количество отрезков, содержащих эту точку.

- **Формат входного файла (input.txt).** Первая строка содержит два неотрицательных целых числа s и p. s - количество отрезков, p - количество точек. Следующие s строк содержат 2 целых числа ai , bi , которые определяют i-ый отрезок [ai, bi]. Последняя строка определяет p целых чисел - точек x1, x2, ..., xp. Ограничения: $1 \le s, p \le 50000$; $-10^8 \le ai \le bi \le 10^8$ для всех $0 \le i < s$; $-10^8 \le xi \le 10^8$ для всех $0 \le j < p$.
- **Формат выходного файла (output.txt).** Выведите p неотрицательных целых чисел k0, k1..., kp−1, где ki - это число отрезков, которые содержат xi .

Листинг кода.

```
from bisect import bisect_left


def segments_parser(lst):
    segments = []
    for l, r in lst:
        segments.append([l, 1])
        segments.append([r, -1])
    return segments

def solution(s, p, lst, points):
    ans = []
    segments = segments_parser(lst)
    segments.sort() # qsort
    segments_for_search = []

    c = 0
    for i in range(2*s):
        x, side = segments[i]
        segments_for_search.append(x)

        segments[i].append(c)
        c += side

    for pt in points:
        idx = bisect_left(segments_for_search, pt)
```

```
        if idx == 2*s or pt < segments[0][0]:
            ans.append(0)
        elif pt == segments[idx][0]:
             ans.append(segments[idx][2] + (1 if segments[idx][1] == -1
else 0))
        else:
            ans.append(segments[idx][2])

    return ans
```

В начале сортируем концы отрезков одним списком по их координате встроенной сортировкой списка *(quicksort)*. После для каждого конца отрезка считаем количество отрезков, с которыми он пересекается. В конце двоичным поиском ищем по концам отрезков точки из условия, находя их взаимное расположение и, следовательно, ответ на задачу.

```
3 2
0 5
-3 2
7 10
1 6
```

```
1
2 0
```

```
.Тест 10 элементов:
Время работы: 1.5100464224815369e-05 секунд
Затрачено памяти: 400 Байт
Тест 1000 элементов:
Время работы: 0.0020663999021053314 секунд
Затрачено памяти: 270.5 Килобайт
Тест 5*10e4 элементов:
Время работы: 0.0963340001180768 секунд
Затрачено памяти: 13.4 Мегабайт
.
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.00001510046422481 секунд	400 Байт
Медиана диапазона значений входных данных из текста	0.0020663999021053314 секунд	270.5 Килобайт

задачи		
Верхняя граница диапазона значений входных данных из текста задачи	0.0963340001180768 секунд	13.4 Мегабайт

Вывод по задаче:
Алгоритм работает корректно: считается количество пересечений. Время выполнения сравнимо с *n*log(n)*, так как присутствует сортировка. Памяти затрачивается относительно мало.

Дополнительные задачи

Задача №5. Индекс Хирша

Для заданного массива целых чисел citations, где каждое из этих чисел - число цитирований i-ой статьи ученого-исследователя, посчитайте индекс Хирша этого ученого.

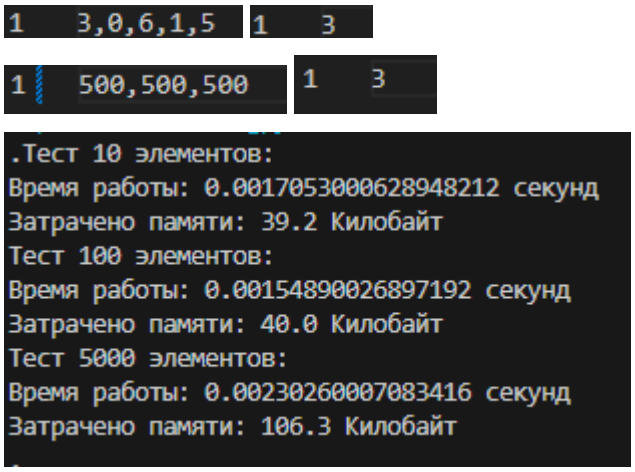
- **Формат входного файла (input.txt).** Одна строка citations, содержащая n целых чисел, по количеству статей ученого (длина citations), разделенных пробелом или запятой.
- **Формат выходного файла (output.txt).** Одно число - индекс Хирша (h-индекс).
- Ограничения: $1 \leq n \leq 5000$, $0 \leq citations[i] \leq 1000$.
- Ограничений по времени (и памяти) не предусмотрено, проверьте максимальный случай при заданных ограничениях на данные, и оцените асимптотическое время.

Листинг кода.

```
def radix_sort(lst):
    freq = [0]*5001
    for i in lst:
        freq[i] += 1
    ans = []
    for n in range(5000, -1, -1):
        if n != 0:
            ans += [n]*freq[n]
    return ans

def solution(citations):
    citations = radix_sort(citations)
    c = 0
    for i in range(len(citations)):
        if citations[i] < c + 1:
            break
        c += 1
    return c
```

Поскольку известные границы значений невелики, используем алгоритм линейной сортировки *radix sort.* После, идем по отсортированному в обратную сторону массиву, пока не встретим количество публикаций

меньшее, чем счётчик (номер текущего элемента с конца), согласно определению индекса Хирша.



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0017141000454314 05 секунд	39.2 Килобайт
Медиана диапазона значений входных данных из текста задачи	0.0015459999904222 79 секунд	40.0 Килобайт
Верхняя граница диапазона значений входных данных из текста задачи	0.0023021700011901 55 секунд	106.3 Килобайт

Вывод по задаче:
Алгоритм работает быстро благодаря линейной сортировке и укладывается в лимит по времени. Также память практически не тратится, поскольку значения небольшие.

Задача №6. Сортировка целых чисел

В этой задаче нужно будет отсортировать много неотрицательных целых чисел. Вам даны два массива, A и B, содержащие соответственно n и m элементов. Числа, которые нужно будет отсортировать, имеют вид Ai · Bj ,

где $1 \le i \le n$ и $1 \le j \le m$. Иными словами, каждый элемент первого массива нужно умножить на каждый элемент второго массива. Пусть из этих чисел получится отсортированная последовательность C длиной $n \cdot m$. Выведите сумму каждого десятого элемента этой последовательности (то есть, C1 + C11 + C21 + ...).

- **Формат входного файла (input.txt).** В первой строке содержатся числа n и m ($1 \le n, m \le 6000$) – размеры массивов. Во второй строке содержится 6 n чисел – элементы массива A. Аналогично, в третьей строке содержится m чисел — элементы массива B. Элементы массива неотрицательны и не превосходят 40000.
- **Формат выходного файла (output.txt).** Выведите одно число — сумму каждого десятого элемента последовательности, полученной сортировкой попарных произведений элементов массивов A и B.
- Ограничение по времени. 2 сек.
- **Ограничение по времени распространяется на сортировку, без учета времени на перемножение. Подумайте, какая сортировка будет эффективнее, сравните на практике.**

Листинг кода.

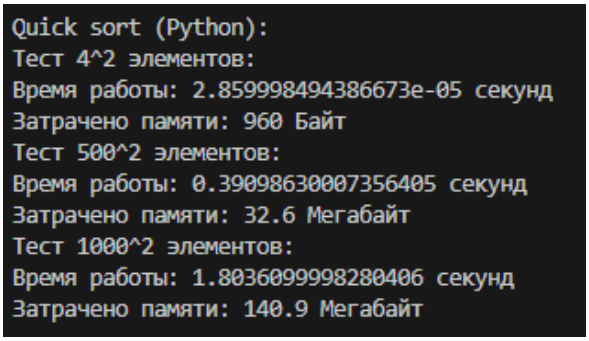
```
import heapq



def radix_sort(lst):
    freq = [0]*40001
    for i in lst:
        freq[i] += 1
    ans = []
    for n in range(40001):
        if n != 0:
            ans += [n]*freq[n]
    lst[:] = ans

def qsort(lst):
    lst.sort()

def heap_algo(n, m, a, b):
    heap = []
    result = []
    visited = set()
```

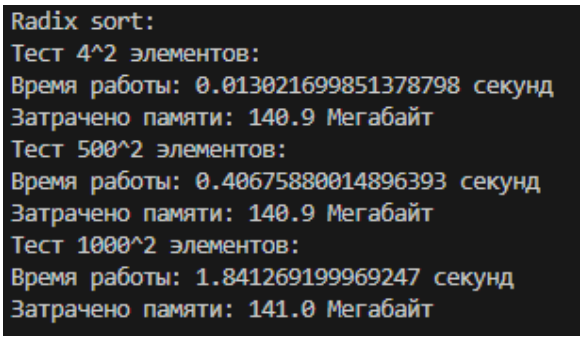
```
    heapq.heappush(heap, (a[0] * b[0], 0, 0))
    visited.add((0, 0))

    while len(result) < n * m:
        product, i, j = heapq.heappop(heap)
        result.append(product)

        if i + 1 < n and (i + 1, j) not in visited: # Push next pairs
            heapq.heappush(heap, (a[i + 1] * b[j], i + 1, j))
            visited.add((i + 1, j))

        if j + 1 < m and (i, j + 1) not in visited:
            heapq.heappush(heap, (a[i] * b[j + 1], i, j + 1))
            visited.add((i, j + 1))

    return result

def solution(n, m, a, b, sort_func):
    sort_func(a)
    sort_func(b)
    res_arr = heap_algo(n, m, a, b)
    return sum(res_arr[i] for i in range(0, n*m, 10))

def main():
    (n, m), a, b = read_file('txtf/input.txt'),
    res = solution(n, m, a, b, radix_sort)
```

Алгоритм сначала сортирует 2 входных массива, чтобы не генерировать большого. Далее программа добавляет в кучу *(heap)* результаты перемножения элементов и текущих индексов так, чтобы вынимать минимальный из них и добавлять в результирующий массив. В конце считается сумма для каждого десятого элемента.

```
4 4
7 1 4 9
2 7 8 11
```

```
51
```





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.013021699851378798 секунд	1.0 Килобайт
Медиана диапазона значений входных данных из текста задачи	0.40675880014896393 секунд	32.6 Мегабайт
Верхняя граница диапазона значений входных данных из текста задачи	1.8036099998280406 секунд	140.9 Мегабайт

Вывод по задаче:
При учете времени на перемножение, которое не должно подпадать под проверку времени выполнения, программа работает эффективно. По сравнению с очевидным решением, задача решается приблизительно в 2 раза быстрее (так как в логарифме n^2 заменяется на n). Также можно заметить, что сортировка *radix sort* лишь приближается по времени выполнения к *quicksort,* но расходует много памяти. Это связано с тем, что встроенная сортировка написана более эффективно и на другом языке, поэтому даже приближенное время выполнения говорит о превосходстве *radix sort* при прочих равных.

Задача №8. K ближайших точек к началу координат

В этой задаче, ваша цель - найти K ближайших точек к началу координат среди данных n точек.

- Цель. Заданы n точек на поверхности, найти K точек, которые находятся ближе к началу координат (0, 0), т.е. имеют наименьшее расстояние до начала координат.
- **Формат входного файла (input.txt).** Первая строка содержит n - общее количество точек на плоскости и через пробел K - количество ближайший точек к началу координат, которые надо найти. Каждая следующая из n строк содержит 2 целых числа xi , yi , определяющие точку (xi , yi). Ограничения: $1 \le n \le 10^5$; $-10^9 \le xi , yi \le 10^9$ - целые числа.
- **Формат выходного файла (output.txt).** Выведите K ближайших точек к началу координат в строчку в квадратных скобках через запятую. Ответ вывести в порядке возрастания расстояния до начала координат. Если оно равно, порядок произвольный
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 256 мб.

Листинг кода.

```
def solution(_, k, points):
    points.sort(key=lambda x: x[0]**2 + x[1]**2)
    return points[:k]

def main():
    (n, k), points = read_multi_lst_file('txtf/input.txt')
    ans = solution(n, k, points)
```

Функция использует встроенную сортировку *(quicksort)* по ключу удалённости точки от центра координат. Далее берется срез отсортированного массива для k элементов и возвращается в качестве ответа.

```
3 2
3 3
5 -1
-2 4
```

```
[3, 3],[-2, 4]
```

```
.Тест 10 элементов:
Время работы: 2.400018274784088e-05 секунд
Затрачено памяти: 160 Байт
Тест 10000 элементов:
Время работы: 0.014638099819421768 секунд
Затрачено памяти: 385.0 Килобайт
Тест 10e5 элементов:
Время работы: 0.15474919974803925 секунд
Затрачено памяти: 3.8 Мегабайт
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0002400000103637576 секунд	160 Байт
Медиана диапазона значений входных данных из текста задачи	0.014638099819421768 секунд	385.0 Килобайт
Верхняя граница диапазона значений входных данных из текста задачи	0.15474919974803925 секунд	3.8 Мегабайт

Вывод по задаче:
Время работы алгоритма сравнимо с *n*log(n),* память практически не расходуется. В этой задаче достаточно ориентироваться на один вычисляемый параметр - расстояние от начала координат, что сильно упрощает алгоритм.

Вывод

Оптимизированные варианты сортировок, особенно те, которые работают за линейное время, могут пригодиться в задачах с известным диапазоном значений, не превышающим определенного порога (в зависимости от вычислительных мощности и памяти). Такие алгоритмы существенно уменьшают время выполнения программы, но, в основном, ценой затрат по памяти и способов применения.