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

Кафедра систем управления и информатики

Отчет по лабораторной работе №1

по дисциплине «Системное программное обеспечение»

2 вариант

Выполнили: студенты гр. R3335

Ал-Наим Рами

Костровская Ольга

Носкова Полина

Преподаватель:

Капитонов А.А.

Санкт-Петербург

2018 г.

Цель работы: закрепление навыков программирования на Python, повторение ранее изученого материала.

Задание 1.

Реализовать алгоритм фасетного поиска для определенной группы объектов Птицы. Реализовать алгоритм не менее чем для 16 объектов. Пользователь может отвечать на вопросы только «да» и «нет».

Решение:

Фасетный поиск— метод доступа к информации, с использованием одновременно нескольких фильтров — фасетов. Для использования фасетного поиска каждая информационная единица классифицируется по нескольким явным характеристикам (фасетам).

Для решения данной задачи нами было выбрано 16 птиц: страус, папугай Ара, воробей, лебедь, утка, кукушка, фламинго, снегирь, петух, орел, индейка, павлин, канарейка, пингвин, дятел и калибри.

Птицы облабают уникальным набором характеристик по которым будет происходить фасетный поиск. Для присвоения каждой из них свойств был написан класс Bird:

```
class Bird:
       def __init__(self, name, size, color, water, wildRussia, sq = None):
              self.name = name
              self.size = size
              self.brightColor = color
              self.waterfowl = water
              self.wildInRussia = wildRussia
              self.specialQuality = sq

       def __str__(self):
              return self.name
```

Каждая птица имеет название(name), размер(size), окрас(brightColor), логическую переменную, отвечающую за водоплавание(waterfowl), логическую переменную, отвечающую за обитание в дикой среде России(wildInRussia), и специальный атрибут, необходимый для определения некоторых птиц(sprecialQuality).

После создания класса заполняем лист init разными птицами (16 штук) с уникальной комбинацией атрибутов. Например, для орла это выглядит следующим образом:

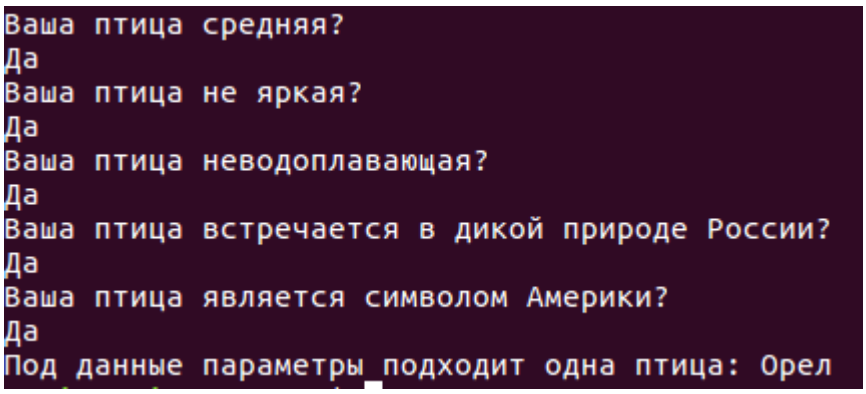
```
eagle = Bird("Орел", size="средняя", color="не яркая",
water="неводоплавающая", wildRussia="встречается в дикой природе России",
sq="является символом Америки")
init.append(eagle)
```

Реализация алгоритма фасетного поиска на примере фильтра по признаку size:

```
# Фильтруем по размеру
if number_of_attributes == 0:

        # заполняем лист possible_attributes возможными значениями параметра
        for bird in init_list:
                if bird.size not in possible_attributes:
                        possible_attributes.append(bird.size)

        # задаем пользователю вопрос до тех пор, пока не останется один возможный атрибут или пользователь ответит Да
        for i in range( len(possible_attributes) - 1 ):
                print( "Ваша птица " + possible_attributes[i] + "?" )
                user_answer = input()
                if user_answer == "Да":
                        index = 0
                        # Оставляем в possible_attributes то значение, которое выбрал пользователь
                        for _ in range(0, len(init_list) ):
                                if init_list[index].size != possible_attributes[i]:
                                        init_list.pop(index)
                                        # Если удалили один элемент, уменьшаем индекс на 1
                                        index -= 1
                                index += 1
                        break
                else:
                        possible_attributes.pop(i)

        # если остается одно возможное значение, удаляем из init_list все неподходящие значения
        if len( possible_attributes ) == 1:
                print("Тогда ваша птица " + possible_attributes[0])
                index = 0
                for _ in range(0, len(init_list) ):
                        if init_list[index].size != possible_attributes[0]:
                                init_list.pop(index)
                                index -= 1
                        index += 1
                        if index == len( init_list ):
                                break
```

В лист possible_attributes сохраняем все возможные значения параметра size, которыми обладают объекты в init_list. Затем пользователю задаются вопросы, и на основании ответов определяется параметр птицы, которую загадал пользователь. Далее init_list фильтруется таким образом, чтобы он содержал только птиц с выбраным значением параметра. Если параметр имеет одно единственное значение, то нет смысла задавать пользователю вопрос, и параметр определяется автоматически. Аналогично список фильтруется остальным параметрам. Программа выполняется до тех пор, пока в отфильтрованном списке не окажется одна птица.

Результат работы программы, если необходимо найти Орла (средняя, неяркая птица, неводоплавающая, обитающая в дикой природе России и являющаяся символом Америки):



Задание 2.

Пользователь задает любое количество чисел с экрана, разделяя их запятыми. Реализовать алгоритм, который распределяет числа на натуральные, целые, рациональные, комплексные, четные, нечетные и простые. Обратите внимание, что цифры могут попадать в несколько категорий.

Решение:

Для начала вспомним определения, необходимые для решения задачи:

Натуральные числа— числа, возникающие естественным образом при счёте (например, 1, 2, 3, 4, 5, 6, 7, 8, 9…).

Целые числа — расширение множества натуральных чисел, получаемое добавлением к нему нуля и отрицательных чисел.

Рациональное число— число, которое можно представить обыкновенной дробью m/n, где числитель m — целое число, а знаменатель n — натуральное число, к примеру 2/3.

Комплексные числа — числа вида a+bi, где a,b — вещественные числа, i — мнимая единица, то есть число, квадрат которого равен -1.

Чётное число — целое число, которое делится на 2 без остатка: …, −4, −2, 0, 2, 4, 6, 8, …

Нечётное число — целое число, которое не делится на 2 без остатка: …, −3, −1, 1, 3, 5, 7, 9, …

Простое число — натуральное (целое положительное) число, имеющее ровно два различных натуральных делителя — единицу и самого себя.

Алгоритм реализации заключается в том, что мы постепенно идем от самого большого множества (комплексные) до самого маленького его подмножества (натуральные), осуществляя проверку на наличие характерных для данного множества черт и записывая числа в соответствующие массивы при их соответствии.

Проверка на четность/нечетность выполняется для массива целых чисел, а проверку на принадлежность числа к простым – для массива натуральных чисел.

Алгоритм определения простого числа:

```
"""Функция принимает в виде аргумента число и возвращает True, если число простое, иначе - False"""
def prime(n):
	"""Если передана едининца - возвращаем 1"""
	if n == 1:
		return False

		"""Достаточно проверить, являются ли все числа от 2 до n/2 (округление в большую сторону), делителями числа n"""
	top_lim = n // 2 + 1
	for den in range(2, top_lim):
		if n % den == 0:
			return False

	return True
```

Результат работы программы:

```
Введите числа через запятую (после запятой пробел, десятичный разделитель - точка, косплексный вид: a+bj):
1.3, 12, 1+7j, 3j, -14, 7, 133, 3.14, -85-933j, 0.60798, 5318008, 2, 5, 13

Комплексные: (1+7j), 3j, (-85-933j)
Рациональные: 1.3, 3.14, 0.60798
Целые: 12, -14, 7, 133, 5318008, 2, 5, 13
Натуральные: 12, 7, 133, 5318008, 2, 5, 13
Четные: 12, -14, 5318008, 2
Нечетные: 7, 133, 5, 13
Простые: 7, 2, 5, 13
```

Задание 3.

Пользователь задает массив натуральных чисел. Реализовать для них алгоритмы сортировки следующими методами: пузырьковый, гномий, блочный, пирамидальный. Проанализировать достоинства и недостатки данных методов.

Решение:

Сортировка пузырьком

Плюсы:

- алгоритм очень лёгкий в реализации;
- устойчивость — устойчивая сортировка не меняет взаимного расположения равных элементов.

Минусы:

- время алгоритма пропорционально квадрату количества элементов (самый медленный способ сортировки). При наихудшем варианте массив будет пройден столько же раз, сколько в нем имеется элементов минус одно значение. Так происходит потому, что в конечном итоге остается только один элемент, который не с чем сравнивать, и последний проход по массиву становится бесполезным действием

Гномья сортировка

Плюсы:

- алгоритм концептуально простой, не требует вложенных циклов;
- устойчивость — устойчивая сортировка не меняет взаимного расположения равных элементов.

Минусы:

- время алгоритма пропорционально квадрату количества элементов (самый медленный способ сортировки). При наихудшем варианте массив будет пройден столько же раз, сколько в нем имеется элементов минус одно значение. На практике алгоритм может работать так же быстро, как и сортировка вставками.

Блочная сортировка

Плюсы:

- устойчивость — устойчивая сортировка не меняет взаимного расположения равных элементов;
- относится к классу быстрых алгоритмов с линейным временем исполнения O(N) (на удачных входных данных).

Минусы:

- сложность алгоритма;
- сильно деградирует при большом количестве мало отличных элементов, или же на неудачной функции получения номера корзины по содержимому элемента

Пирамидальная сортировка

Плюсы:

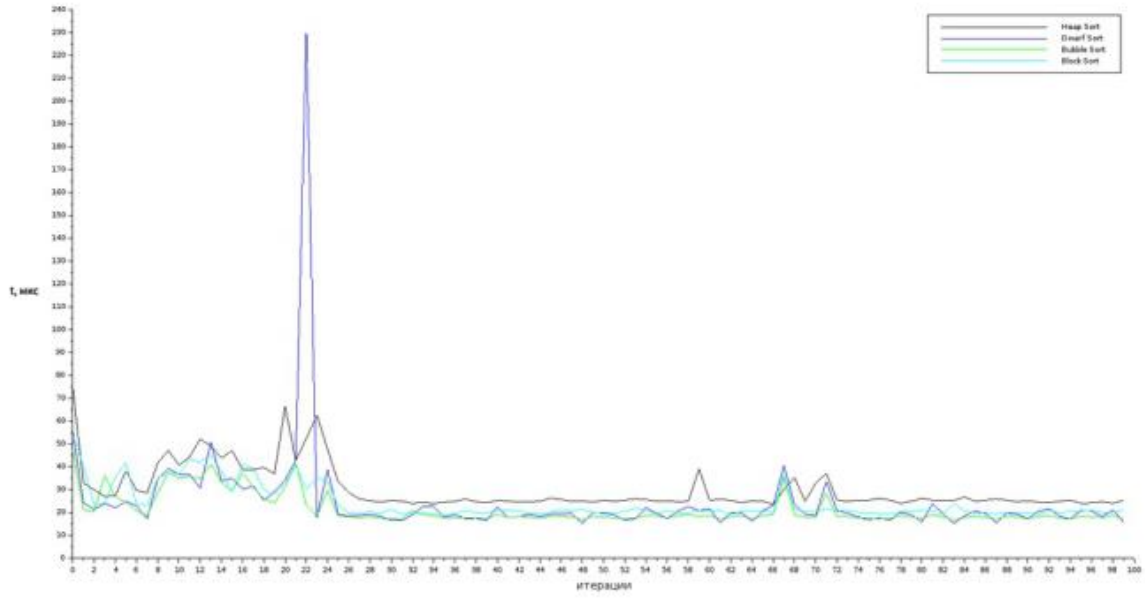
- имеет доказанную оценку худшего случая O(n*log n);
- сортирует на месте, то есть требует всего O(1) дополнительной памяти;
- не подвержена деградации на неудачных данных.

Минусы:

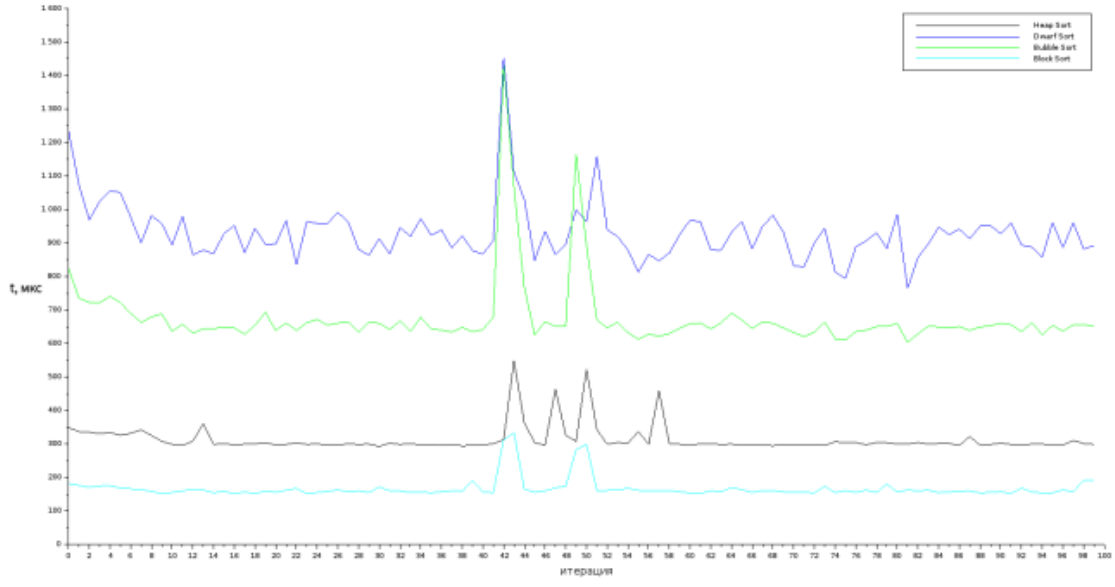
- сложен в реализации;
- алгоритм неустойчив;
- на почти отсортированных массивах работает столь же долго, как и на хаотических данных;
- Из-за сложности алгоритма выигрыш получается только на больших n.

Каждый из алгоритмов был использован для сортировки 100 случайный массивов длиной 10(а) и 100(б) элементов.

а) Среднее время сортировки массива из 10 элементов (в мкс):
Пирамидальная: 26.19
Гномья: 19.99
Пузырьковая: 19.16
Блочная: 21.47



б) Среднее время сортировки массива из 100 элементов (в мкс):
Пирамидальная: 313.71
Гномья: 931.27
Пузырьковая: 675.67
Блочная: 167.45



Как видно из графиков, при массивах меньшей длины пузырьковая и гномья сортировки выполняются быстрее, чем пирамидальная и блочная, однако при массивах большей длины пирамидальная и блочные сортировки работают быстрее. Это можеи быть вызвано тем, что рекурсивный вызов функции при пирамидальной сортировке (и вызов сортировки пузырьком в блочной сортировке) для коротких массивов занимает большое

количество времени, однако при обработке больших массивов это выгоднее по сравнению с итерационными обходами массива в гномьей и пузырьковой сортировках.

Задание 4.

С А третьекурсниками часто происходит событие B. Зная вероятность, что данное событие происходит с ними N раз в C дней (N вводится пользователем при запуске программы), определите вероятность того, что за D дней данная ситуация произойдет только с третьекурсником E. Реализовать алгоритм для решения данной задачи (см. Таблица 2 и Теорему Байеса).

Вариант	**A**	**B**	**C**	**D**	**E**
2	Никита Сережа Настя	Оползень	14	200	Никита

Решение:

При решении задачи на теорию вероятности используем базовую формулу (1 - (1-P1) ^ (D / C)) * (1-P2) ^ (D / C) * (1-P3) ^ (D / C), где Pi – введенное нами значение вероятности, а C и D - уже известные нам параметры. Не забываем поставить ограничение на вводимую вероятность – она не должна быть больше 1 и меньше 0.

Результат работы программы:

```
Введите три вероятности для страшного оползня: для Никиты, Сережи и Насти.
P.S.Вероятность может принимать значения от 0 до 1
Никита: 0.1
Сережа: 0.4
Настя: 0.69
Вероятность того, что за 200 дней оползень произойдет только с Никитой: 0.774
```

Вывод: В данной лабораторной работе все поставленные перед нами задачи были успешно реализованы. Мы вспомнили основные принципы программирования на Python, создавали собственный класс, использовали циклы, условные операторы, работали с числами и строками.