Національний технічний університет України «КПІ ім. Ігоря Сікорського»

Факультет Інформатики та Обчислювальної Техніки

Кафедра Автоматизованих Систем Обробки Інформації та Управління

Лабораторна робота № 2

з дисципліни «Теорія алгоритмів»

на тему:

**"Cортування включенням"**

Виконав:

студент гр. ІС-02

Плостак Ілля

Викладач:

Cт. Вик. Новікова П. А.

Київ – 2021

**1. Завдання**

1. Реалізувати алгоритми сортування методами бульбашки ( функцію bubble\_sort, improve\_bubble\_sort) та включення (функцію insertion\_sort). При чому під час роботи алгоритму повинні підраховуватись порівняння елементів, які виконує кожен з методів.
2. Провести тестування алгоритмів на різних наборах даних. Для тестування використовується функція generate\_data. Тестування проводиться для розмірів даних [10, 100, 1000, 10000] та трьох типів вхідних даних (випадково згенеровані вхідні дані, найкращі та найгірші вхідні дані). Порівняти отримані результати тестування.
3. Знайти спосіб покращити поведінку метода бульбашки, щоб він використовував меншу кількість операцій. Реалізувати знайдений спосіб та дійти висновку на скільки це вплинуло на якість роботи алгоритму.

**2. Програмний код:**

[main.py](https://github.com/feedblackg44/kpilabs2/blob/master/TA/Lab2/main.py):

import numpy as np  
import random  
  
import func  
import plot\_func as pf  
  
sizes = [10, 100, 1000, 10000]  
types = ["random", "best", "worst"]  
data\_plot = {'random': {'bubble':{}, 'insertion':{}, 'bubble\_impr':{}},   
 'best': {'bubble':{}, 'insertion':{}, 'bubble\_impr':{}},  
 'worst': {'bubble':{}, 'insertion':{}, 'bubble\_impr':{}}}  
  
for n in sizes:  
 print("\nDATA SIZE: ", n)  
 for gen\_type in types:  
 print("\n\tDATA TYPE:", gen\_type)  
 data = func.generate\_data(n, gen\_type)  
  
 data\_bubble = np.copy(data)  
 bubble\_op\_count = func.bubble\_sort(data\_bubble)  
 print ("\tBubble sort operation count:", int(bubble\_op\_count))  
 data\_plot[gen\_type]['bubble'][n] = bubble\_op\_count  
  
 data\_bubble\_impr = np.copy(data)  
 bubble\_impr\_op\_count = func.bubble\_impr\_sort(data\_bubble\_impr)  
 print("\tImproved bubble sort operation count:", int(bubble\_impr\_op\_count))  
 data\_plot[gen\_type]['bubble\_impr'][n] = bubble\_impr\_op\_count  
  
 data\_insertion = np.copy(data)  
 insertion\_op\_count = func.insertion\_sort(data\_insertion)  
 print("\tInsertion sort operation count:", int(insertion\_op\_count))  
 data\_plot[gen\_type]['insertion'][n] = insertion\_op\_count  
  
pf.plot\_data(data\_plot, logarithmic=True, oneplot=False)

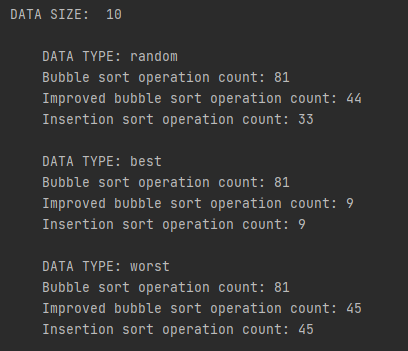
[func.py](https://github.com/feedblackg44/kpilabs2/blob/master/TA/Lab2/func.py):

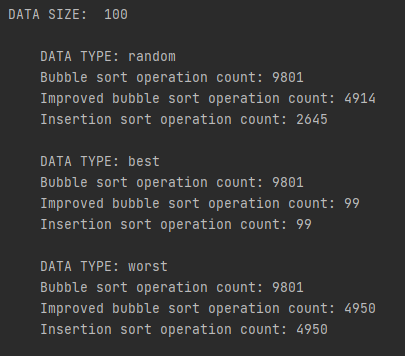
import random  
  
def generate\_data(n, gen\_type="random"):  
 if gen\_type=="best":  
 a = [i+1 for i in range(n)]  
 return a  
 elif gen\_type=="worst":  
 a = [i+1 for i in reversed(range(n))]  
 return a  
 else:  
 a = [i+1 for i in range(n)]  
 random.shuffle(a)  
 return a  
  
def bubble\_sort(data):  
 counter = 0  
 for i in range(0, len(data) - 1, 1): # Проходим по всем элементам массива в колчиестве раз равным количеству элементов  
 for j in range(0, len(data) - 1, 1): # Проходим по всем элементам массива  
 counter += 1  
 if (data[j] > data[j + 1]): # Проверка, если текщуий элемент больше следующего  
 data[j + 1], data[j] = data[j], data[j + 1] # Переставляем элементы в таком случае  
 return counter  
  
def bubble\_impr\_sort(data):  
 counter = 0  
 for i in range(0, len(data) - 1, 1): # Проходим по всем элементам массива в колчиестве раз равным количеству элементов  
 swap = False # Флаг для проверки, не перемещались ли элементы  
 for j in range(0, len(data) - i - 1, 1): # Проходим по всем оставшимся элементам массива  
 counter += 1  
 if (data[j] > data[j + 1]): # Проверка, если текщуий элемент больше следующего  
 data[j + 1], data[j] = data[j], data[j + 1] # Переставляем элементы в таком случае  
 swap = True  
 if (swap == False): # Если элементы не перемещались - выходим из цикла, т.к. массив сортирован  
 break  
 return counter  
  
  
def insertion\_sort(data):  
 counter = 0  
 for j in range(1, len(data)): # Проходим все элементы массива, кроме нулевого  
 shift = False # Переменная для проверки сдвига  
 key = data[j] # Ключ - текущий элемент  
 i = j - 1 # Индекс предыдущего элемента  
 while i >= 0 and data[i] > key: # Перебираем элементы от ключа в начало, пока ключ меньше текущего элемента  
 data[i + 1] = data[i] # Присваиваем следующему элементу значение текущего  
 i -= 1  
 counter += 1  
 shift = True  
 if i >= 0: # Если не прошли весь массив увеличиваем каунтер на одно сравнение  
 counter += 1  
 if shift: # Если были сдвиги  
 data[i + 1] = key # Переставляем ключ на нужную позицию  
 return counter

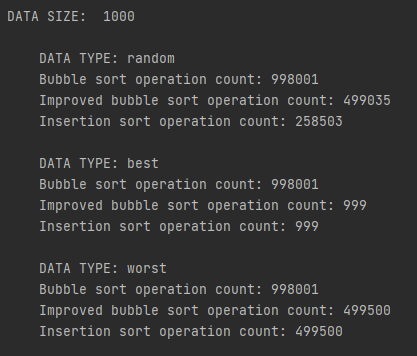
[plot\_func.py](https://github.com/feedblackg44/kpilabs2/blob/master/TA/Lab2/plot_func.py):

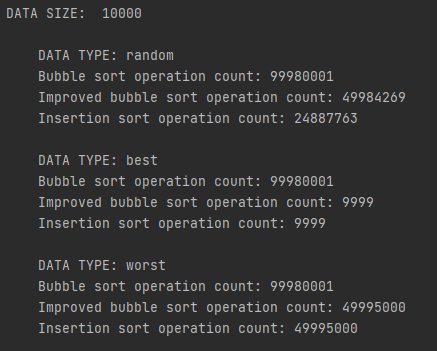
import collections  
from pylab import figure, show  
from math import log  
  
def plot\_data(data, logarithmic=False, oneplot=False):  
 fig = figure(1)  
 num = len(data)  
 colors = ['r','b','g']  
 markers = ['s','o','x']  
 lines = ['-','--',':']  
 if oneplot==True:  
 ax = fig.add\_subplot(111)  
 ax.grid(True)  
 i = -1  
 line\_titles = []  
 x\_max = y\_max = 0  
 for label, value in data.items():  
 i += 1  
 j = -1  
 for sort\_type, points in value.items():  
 j += 1  
 od\_points = collections.OrderedDict(sorted(points.items()))  
 if logarithmic:  
 xs = [(x>0 and log(x,10) or 0) for x in od\_points.keys()]  
 ys = [(y>0 and log(y,10) or 0) for y in od\_points.values()]  
 else:  
 xs = list(od\_points.keys())  
 ys = list(od\_points.values())  
 xs.insert(0,0)  
 x\_max = max(x\_max, max(xs))  
 ys.insert(0,0)  
 y\_max = max(y\_max, max(ys))  
 ax.plot(xs, ys, colors[j%num]+markers[j%num]+lines[i%num], label=sort\_type )  
 line\_titles.append(sort\_type+' '+label)  
 ax.set\_xlim( (0, x\_max\*1.1) )  
 ax.set\_ylim( (0, y\_max\*1.1) )  
 ax.legend(line\_titles, loc=4)  
 else:  
 i = 0  
 for label, value in data.items():  
 i += 1  
 ax = fig.add\_subplot(num,1,i)  
 ax.grid(True)  
 ax.set\_title(label)  
 j = -1  
 x\_max = y\_max = 0  
 for sort\_type, points in value.items():  
 j += 1  
 od\_points = collections.OrderedDict(sorted(points.items()))  
 if logarithmic:  
 xs = [log(x,10) for x in od\_points.keys()]  
 ys = [log(y,10) for y in od\_points.values()]  
 else:  
 xs = od\_points.keys()  
 ys = od\_points.values()  
 xs.insert(0,0)  
 x\_max = max(x\_max, max(xs))  
 ys.insert(0,0)  
 y\_max = max(y\_max, max(ys))  
 ax.plot(xs, ys, colors[j%num]+markers[j%num]+'-', label=sort\_type )  
 ax.set\_xlim( (0, x\_max\*1.1) )  
 ax.set\_ylim( (0, y\_max\*1.1) )  
 ax.legend(loc=4)  
 show()

**3. Результат запуску програми:**

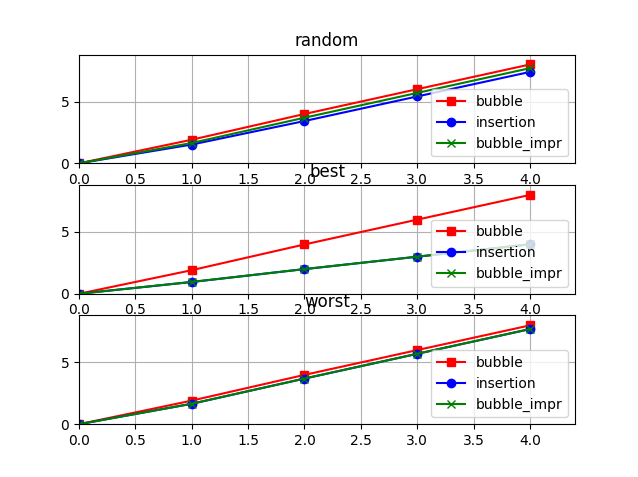








**4. Графіки:**



**5. Висновок:**

В результаті цієї лабораторної роботи ми порівняли три методи сортування: бульбашку, покращену бульбашку та сортування включенням. Найдовшим з них є сортування бульбашкою, яке у всіх випадках має однаковий час виконання незалежно від того, наскільки дані вже сортовані. На другому місці покращений алгоритм сортування бульбашкою. По перше, в найгіршому випадку він виконується вдвічі швидше ніж звичайний алгоритм, тому що для кожної наступної ітерації внутрішній for виконується на один раз менше ніж для попередньої ітерації. Також, в покращеній бульбашці з`являється флаг для перевірки, чи сортований масив вже, чи ні. Та якщо він вже сортований, то ми виходимо із цикла достроково. Це забезпечує залежність часу сортування від того, наскільки дані вже сортовані. На першому місці – сортування включенням. Головна відмінність цього методу сортування від бульбашки це те, що немає операцій перестановки елементів. Замість цього в даному алгоритмі елемент за одну ітерацію одразу потрапляє на своє місце. Ми просто порівнюємо поточний елемент з кожним перед ним та ставимо його туди, де попередній елемент не буде більше за поточний, який ми називаємо «ключовим». А усі інші елементи, починаючи з фінальної позиції ключового елемента закінчуючи старою його позицією не включно, зсуваємо на один, щоб заповнити утворену «дірку». Такий алгоритм у найгіршому та найліпшому випадку співпадає з покращеною бульбашкою, але на проміжку, коли ми беремо випадкові значення, сортування включенням виконується за менший час ніж покращена бульбашка. І таким чином серед перерахованих вище методів сортування найекономнішим є саме алгоритм сортування включенням.