Санкт-Петербургский Национальный Исследовательский Университет

Информационных Технологий, Механики и Оптики

Факультет информационных технологий

Лабораторная работа №3

Выполнили:

Лазуренко А.В., Маатук А.

Проверил

Мусаев А.А.

Санкт-Петербург,

2023

ЗАДАНИЕ 1

Для выполнения задачи импортировали следующие модули

import random  
import timeit

*Рисунок 1 – Импорт модулей*

Далее реализовали алгоритм быстрой сортировки на основе изученных данных.

def quick\_sort(arr):  
 if len(arr) <= 1:  
 return arr  
 pivot = arr[len(arr) // 2]  
 left = [x for x in arr if x < pivot]  
 middle = [x for x in arr if x == pivot]  
 right = [x for x in arr if x > pivot]  
 return quick\_sort(left) + middle + quick\_sort(right)

*Рисунок 2 – Быстрая сортировка*

Сложность алгоритма составляет O(n\*log(n)) (средняя), O(n) (лучшая)

Также реализовали метод сортировки «расческой»

def comb\_sort(arr):  
 gap = len(arr)  
 shrink = 1.3  
 sorted = False  
  
 while not sorted:  
 gap = int(gap / shrink)  
 if gap <= 1:  
 gap = 1  
 sorted = True  
 i = 0  
 while i + gap < len(arr):  
 if arr[i] > arr[i + gap]:  
 arr[i], arr[i + gap] = arr[i + gap], arr[i]  
 sorted = False  
 i += 1  
  
 return arr

*Рисунок 3 – Сортировка «расческой»*

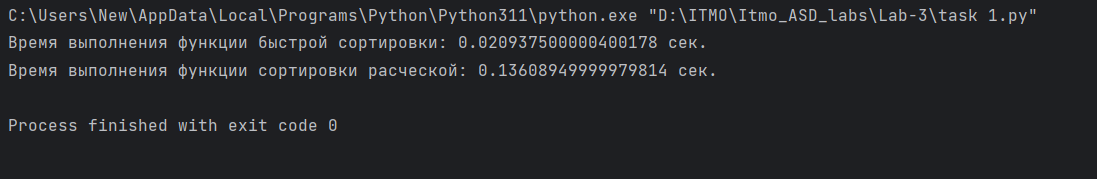
Сложность алгоритма составляет O(n\*log(n)) (лучшая), O(n^2/2^p) (среднее)

Затем создали массив из случайных чисел и сравнили скорость работы алгоритмов на этом массиве с помощью модуля timeit

# Генерируем случайный массив для сортировки  
arr = [random.randint(0, 1000) for \_ in range(10000)]  
  
# Замеряем время выполнения функции быстрой сортировки  
quick\_sort\_time = timeit.timeit(lambda: quick\_sort(arr), number=1)  
  
# Замеряем время выполнения функции сортировки расческой  
comb\_sort\_time = timeit.timeit(lambda: comb\_sort(arr.copy()), number=1)  
  
print(f'Время выполнения функции быстрой сортировки: {quick\_sort\_time} сек.')  
print(f'Время выполнения функции сортировки расческой: {comb\_sort\_time} сек.')

*Рисунок 4 – Алгоритм сравнения результатов*

Вывод программы представлен на рисунке 5



*Рисунок 5 – Вывод программы*

В результате тестов можем сделать вывод, что «Быстрая» сортировка намного эффективнее сортировки «расческой».

ЗАДАНИЕ 2

Для выполнения задания изучили 3 алгоритма сортировки – «Блочную», «Пирамидальную» и сортировку «Слиянием».

Далее представлена реализация алгоритмов.

#Блочная сортировка  
def floor\_up(i):  
 return int(i//1)  
def bubble\_sort(arr):  
 n = len(arr)  
  
 for i in range(n):  
 for j in range(n - i - 1):  
 if arr[j] > arr[j + 1]:  
 temp = arr[j]  
 arr[j] = arr[j + 1]  
 arr[j + 1] = temp  
 return arr  
def bucket\_sort(arr):  
 bckts=[[] for i in range (len(arr))]  
 minval=arr[0]  
 maxval=arr[0]  
 for i in arr:  
 if i<minval: minval=i  
 if i>maxval: maxval=i  
  
 numrange=maxval-minval  
  
 for i in arr:  
 bcktidx=floor\_up((i-minval) / numrange \* (len(bckts)-1))  
 bckts[bcktidx].append(i)  
  
 for i in range(len(bckts)):  
 bckts[i]=bubble\_sort(bckts[i])  
  
  
 res=[]  
 for i in range(len(arr)):  
 for j in range(len(bckts[i])):  
 res.append(bckts[i][j])  
 return res

*Рисунок 6 – Блочная сортировка*

Так как блочная сортировка требует дополнительной сортировки элементов в»блоках» (корзинах), то используем для решения этой задачи простейший алгоритм – сортировку пузырьком. Блочная сортировка имеет недостаток в том, что при сортировке очень близких друг к другу значений, почти все элементы отправляются в одну корзину, из-за чего увеличивается время ее вторичной сортировки. Сложность алгоритма составляет O(n) – лучшее, O(n^2) – худшее.

#Пирамидальная сортировка  
def swap(arr, i, j):  
 res=arr  
 res[i], res[j] = res[j], res[i]  
 return res  
  
def sift\_down(arr, i, upper):  
 res=arr  
 while True:  
 l, r = i\*2+1, i\*2+2  
  
 if max(l, r)<upper: #this indexes is exist in now arr  
 if res[i]>=max(res[l], res[r]): break #branch in tree is sorted  
 elif res[l]>res[r]:  
 res=swap(res, i, l)  
 i = l  
 else:  
 res = swap(res, i, r)  
 i = r  
  
 elif l<upper: #only left element exist  
 if res[l]>res[i]:  
 res=swap(res, i, l)  
 i = l  
 else:  
 break  
  
 elif r<upper: # only right element exist  
 if res[l] > res[i]:  
 res = swap(res, i, l)  
 i = l  
 else:  
 break  
  
 else: break #no children elements in this tree  
 return res  
def heap\_sort(arr):  
 res=arr  
 for i in range((len(res)-2)//2, -1, -1):  
 res=sift\_down(res, i, len(res))  
  
 for end in range(len(res)-1, 0, -1):  
 res=swap(res, 0, end)  
 res=sift\_down(res, 0, end)  
  
  
 return res

*Рисунок 7 – Пирамидальная сортировка*

Пирамидальная сортировка в данной форме имеет фиксированную сложность - O(n\*log(n)), что является ее достоинством.

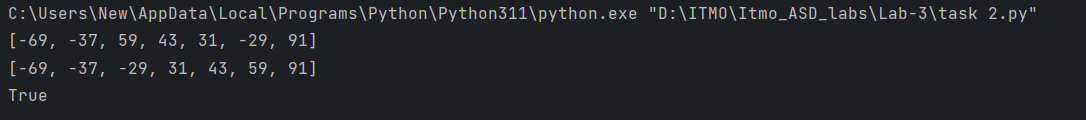
#Сортировка слиянием  
def merge\_sort(arr):  
 if len(arr) <= 1:  
 return arr  
# find the middle index(العثور على الفهرس الأوسط)  
 mid = len(arr) // 2 # integer  
 left = merge\_sort(arr[:mid]) # left part  
 right = merge\_sort(arr[mid:]) # right part  
 return merge(left, right)  
  
  
def merge(left, right):  
 result = []  
 i = j = 0  
  
 while i < len(left) and j < len(right):  
 if left[i] < right[j]:  
 result.append(left[i])  
 i += 1  
 else:  
 result.append(right[j])  
 j += 1  
  
 while i < len(left):  
 result.append(left[i])  
 i += 1  
  
 while j < len(right):  
 result.append(right[j])  
 j += 1  
  
 return result

*Рисунок 8 – Сортировка слиянием*

Сортировка слиянием имеет фиксированную сложность - O(n\*log(n)), что является ее достоинством.

Вывод программ, следующий (аналогичен для всех вариантов.

#Проверка  
arr = [random.randint(-100, 100) for \_ in range(7)]  
print(arr)  
  
mysorted = heap\_sort(arr) #choose your sort mode  
  
print(mysorted)  
if mysorted==sorted(arr): print("True")  
else: print("False")



*Рисунок 9 – Проверка и вывод программы*

ЗАДАНИЕ 3

Еще раз подведем итоги по сложности всех алгоритмов:

* Быстрая: O(n\*log(n)) (средняя)
* Расческой: O(n\*log(n)) (лучшая)
* Блочная: O(n) (лучшая), O(n^2) (худшая)
* Пирамидальная: O(n\*log(n)) (фиксированная)
* Слиянием: O(n\*log(n)) (фиксированная)

Если заранее известна информация о входных данных, и они хорошо совместимы с Блочной сортировкой, то лучше выбрать её для решения. В остальных же случаях удобно будет выбрать Пирамидальную сортировку или сортировку слиянием, так как они имеют фиксированную сложность, примерно равную средней сложности остальных алгоритмов.