**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ,**

**СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА» (СПбГУТ)**

Факультет: Инфокоммуникационных сетей и систем Кафедра: Защищенных систем связи

Дисциплина: Защищенные операционные системы

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5

Методы сортировки

*(тема отчета)*

Направление/специальность подготовки

10.03.01 Информационная безопасность

*(код и наименование направления/специальности)*

Студент:

Емельянов Н.Р. ИКБ-32

*(Ф.И.О., № группы) (подпись)*

Михайлов И.Д. ИКБ-32

*(Ф.И.О., № группы) (подпись)*

Преподаватель:

Асс. Смирнов Д.Н.

*(Должность, Ф.И.О. преподавателя) (подпись)*

Санкт-Петербург 2024

**Цель**

Исследовать эффективности различных методов сортировки, используя язык программирования Python.

**Задачи**

Основные задачи на лабораторную работу:

1. Реализовать алгоритм сортировки пузырьком и оценить его производительность на случайных данных различного размера;
2. Написать сортировку выбором и проверить ее эффективность на упорядоченных, обратно упорядоченных и случайных данных;
3. Создать алгоритм сортировки вставками, и проверить ее работу на списках различной длины;
4. Создать функцию для сортировки слиянием и протестировать ее на больших объемах данных;
5. Реализовать алгоритм быстрой сортировки и проверить его производительность на случайных и повторяющихся данных;
6. Организовать сортировку Timsort. Протестировать его на различных типах данных, включая данные с повторяющимися значениями;
7. Написать сортировку кучей и провести эксперименты с ее производительностью;
8. Внедрить в код алгоритм блочной сортировки и проверить ее на эффективность, используя различные размеры блоков;
9. Создать программу для сортировки радиксом и провести эксперименты с разными системами счисления и разрядами;
10. Сравнить производительность каждой сортировки.

**Ход выполнения**

Шаги выполнения лабораторной работы:

1. Импортируем библиотеки: random, time;
2. Создаем функцию is\_number, которая проверяет, является ли входная строка числом;
3. Инициализируем тестовый массив при помощи функции generate\_list();
4. Реализуем bubble sort при помощи функции sort\_bubble():
   1. Принимается список massiv\_to\_sort, который нужно отсортировать;
   2. Создается копия исходного списка massiv\_to\_sort под названием sorted\_massiv;
   3. Получается длина списка N;
   4. Процесс сортировки пузырьком:
   5. Внешний цикл for i in range(N - 1) проходит по всем элементам списка, за исключением последнего, так как после каждой итерации внутреннего цикла самый большой элемент "всплывает" на своё место;
   6. Внутренний цикл for j in range(N - 1 - i) сравнивает каждую пару соседних элементов и, если порядок неправильный (меньший элемент стоит после большего), меняет их местами;
   7. Таким образом, на каждой итерации внешнего цикла самый большой элемент "всплывает" на своё место в конце списка;
   8. После завершения всех итераций сортировки возвращается копия отсортированного списка sorted\_massiv;
5. Реализуем selection sort при помощи функции sort\_selection():
   1. Процесс сортировки выбором:
   2. Внешний цикл for i in range(n - 1) проходит по всем элементам списка, кроме последнего;
   3. На каждой итерации внешнего цикла выбирается индекс min, который указывает на текущий минимальный элемент (предполагаем, что он находится в позиции i);
   4. Во внутреннем цикле for j in range(i + 1, n) происходит поиск минимального элемента среди оставшихся элементов списка;
   5. Если находится элемент меньше, чем элемент в позиции min, то обновляется индекс min;
   6. После завершения внутреннего цикла текущий минимальный элемент меняется местами с элементом в позиции I;
   7. После завершения всех итераций сортировки возвращается копия отсортированного списка sorted\_massiv;
6. Создаем insertion sort при помощи функции sort\_insertion():
   1. Создается копия входного списка massiv\_to\_sort под названием sorted\_massiv;
   2. Для каждого элемента, начиная со второго (индекс 1) до конца списка:

Сравнивается текущий элемент с предыдущими элементами в отсортированной части списка;

Если текущий элемент меньше предыдущего, то они меняются местами;

* 1. Таким образом, каждый элемент "вставляется" на свое место в отсортированную часть списка;
  2. Возвращается отсортированный список sorted\_massiv;

1. Пишем merge sort при помощи функции sort\_merge():
   1. Принимается список massiv\_to\_sort, который нужно отсортировать;
   2. Получается длина списка size;
   3. Если длина списка больше 1, то происходит разделение списка на две части:

Середина списка вычисляется как middle = size // 2;

Левая половина списка сохраняется в переменной left\_arr;

Правая половина списка сохраняется в переменной right\_arr;

* 1. Рекурсивно вызывается функция sort\_merge для левой и правой половин списка;
  2. Затем выполняется процесс слияния отсортированных половин:

Используются индексы p, q, и r для отслеживания текущих позиций в левой, правой и итоговой частях списка соответственно;

Пока индексы p и q не достигнут конца своих соответствующих частей, сравниваются элементы и записывается минимальный из них в итоговую часть;

После этого увеличиваются соответствующие индексы;

После завершения цикла оставшиеся элементы из каждой части списка добавляются в итоговую часть;

* 1. Возвращается копия отсортированного списка massiv\_to\_sort;

1. Создаем алгоритм quick sort при помощи функции sort\_quick();
   1. Принимается список massiv\_to\_sort, который нужно отсортировать;
   2. Если длина списка меньше или равна 1, то возвращается сам список (так как он уже отсортирован);
   3. Иначе:

Выбирается случайный элемент q из списка massiv\_to\_sort;

Создаются три пустых списка: s\_nums для элементов меньше q, m\_nums для элементов больше q, и e\_nums для элементов равных q;

Проходится по всем элементам списка massiv\_to\_sort и распределяются по соответствующим спискам (s\_nums, m\_nums, e\_nums) в зависимости от их значения относительно q;

* 1. Рекурсивно вызывается функция sort\_quick для списков s\_nums и m\_nums;
  2. Возвращается результат отсортированных списков s\_nums, e\_nums и m\_nums;

1. Реализуем timsort при помощи функции sort\_timsort():
   1. Sorted в стандартной библиотеке работает по тому же принципу, что и timsort, поэтому функция возвращает sorted(massiv\_to\_sort);
2. Внедряем в код heap sort при помощи функций heapify и sort\_heap
3. Создаем функцию sort\_counting для реализации сортировки подсчётом:
   1. На первом этапе функция находит максимальное значение M во входном массиве input\_array;
   2. Затем создается массив count\_array длиной (M + 1), который будет использоваться для подсчета количества каждого элемента;
   3. Происходит первый проход по входному массиву input\_array, где для каждого элемента увеличивается соответствующий счетчик в count\_array;
   4. Далее происходит накопление счетчиков, где каждый элемент i в count\_array равен сумме всех предыдущих элементов и самого элемента I;
   5. Создается массив output\_array длиной равной входному массиву, который будет содержать отсортированные элементы;
   6. Происходит второй проход по входному массиву в обратном порядке, где для каждого элемента input\_array[i] его позиция в отсортированном массиве определяется с помощью count\_array[input\_array[i]] - 1. После этого значение в output\_array на этой позиции устанавливается равным input\_array[i], и счетчик в count\_array уменьшается на 1;
   7. После завершения всех итераций возвращается отсортированный массив output\_array;
4. Реализуем bucket sort при помощи функции sort\_bucket():
   1. На первом этапе функция находит максимальное значение max\_value во входном массиве massiv\_to\_sort;
   2. Затем определяется размер корзины size, который равен отношению max\_value к длине массива massiv\_to\_sort;
   3. Создается список buckets\_list, содержащий пустые корзины, количество которых равно длине массива;
   4. Происходит первый проход по входному массиву massiv\_to\_sort, где каждый элемент размещается в соответствующей корзине на основе значения j = int(massiv\_to\_sort[i] / size);
   5. Далее для каждой непустой корзины вызывается функция sort\_insertion, которая сортирует элементы внутри корзины (по методу сортировки вставками);
   6. Затем происходит объединение всех отсортированных корзин в итоговый отсортированный массив final\_output;
   7. Наконец, функция возвращает отсортированный массив final\_output;
5. Пишем radix sort при помощи функций countingSortForRadix() и sort\_radix():
   1. Функция countingSortForRadix() сначала создает массив countArray размером 10 для подсчета количества элементов в каждой корзине, затем проходит по входному массиву, увеличивая соответствующий счетчик в countArray. После этого происходит накопление количества элементов в каждой корзине, и элементы перемещаются в итоговый массив outputArray на основе значений разрядов и счетчиков;
   2. Функция sort\_radix() принимает массив для сортировки massiv\_to\_sort, определяет максимальный элемент в массиве и количество разрядов чисел. Затем она вызывает countingSortForRadix() для каждого разряда, начиная с младшего разряда, и возвращает отсортированный массив;
6. Реализуем взаимодействие с пользователем, информируя о сгенерированном массиве чисел и предоставляя выбор типа сортировки;
7. Анализируем время выполнения кода в зависимости от выбранного типа сортировки и формируем вывод из полученных данных.

**Листинг программы**

from random import randint

import time

import random

# названия функций сортировки начинаем с sort

def is\_number(input\_str):

try:

int(input\_str)

return True

except ValueError:

return False

def generate\_list(numbers): # инициализируем тестовый массив

testmassiv = []

for k in range(numbers):

testmassiv.append(randint(-1 \* numbers, numbers))

return testmassiv

def sort\_bubble(massiv\_to\_sort):

sorted\_massiv = massiv\_to\_sort.copy()

N = len(sorted\_massiv)

for i in range(N - 1):

for j in range(N - 1 - i):

if sorted\_massiv[j] > sorted\_massiv[j + 1]:

sorted\_massiv[j], sorted\_massiv[j + 1] = (

sorted\_massiv[j + 1],

sorted\_massiv[j],

)

return sorted\_massiv.copy()

def sort\_selection(massiv\_to\_sort):

sorted\_massiv = massiv\_to\_sort.copy()

n = len(sorted\_massiv)

for i in range(n - 1):

min = i

for j in range(i + 1, n):

if sorted\_massiv[j] < sorted\_massiv[min]:

min = j

sorted\_massiv[i], sorted\_massiv[min] = sorted\_massiv[min], sorted\_massiv[i]

return sorted\_massiv.copy()

def sort\_insertion(massiv\_to\_sort):

sorted\_massiv = massiv\_to\_sort.copy()

for i in range(1, len(sorted\_massiv)):

for j in range(i - 1, -1, -1):

if sorted\_massiv[j] > sorted\_massiv[j + 1]:

sorted\_massiv[j], sorted\_massiv[j + 1] = (

sorted\_massiv[j + 1],

sorted\_massiv[j],

)

return sorted\_massiv.copy()

def sort\_merge(massiv\_to\_sort):

size = len(massiv\_to\_sort)

if size > 1:

middle = size // 2

left\_arr = massiv\_to\_sort[:middle]

right\_arr = massiv\_to\_sort[middle:]

sort\_merge(left\_arr)

sort\_merge(right\_arr)

p = 0

q = 0

r = 0

left\_size = len(left\_arr)

right\_size = len(right\_arr)

while p < left\_size and q < right\_size:

if left\_arr[p] < right\_arr[q]:

massiv\_to\_sort[r] = left\_arr[p]

p += 1

else:

massiv\_to\_sort[r] = right\_arr[q]

q += 1

r += 1

while p < left\_size:

massiv\_to\_sort[r] = left\_arr[p]

p += 1

r += 1

while q < right\_size:

massiv\_to\_sort[r] = right\_arr[q]

q += 1

r += 1

return massiv\_to\_sort.copy()

def sort\_quick(massiv\_to\_sort):

if len(massiv\_to\_sort) <= 1:

return massiv\_to\_sort

else:

q = random.choice(massiv\_to\_sort)

s\_nums = []

m\_nums = []

e\_nums = []

for n in massiv\_to\_sort:

if n < q:

s\_nums.append(n)

elif n > q:

m\_nums.append(n)

else:

e\_nums.append(n)

return sort\_quick(s\_nums) + e\_nums + sort\_quick(m\_nums)

def sort\_timsort(massiv\_to\_sort):

return sorted(massiv\_to\_sort)

def heapify(arr, n, i):

largest = i

l = 2 \* i + 1

r = 2 \* i + 2

if l < n and arr[i] < arr[l]:

largest = l

if r < n and arr[largest] < arr[r]:

largest = r

if largest != i:

(arr[i], arr[largest]) = (arr[largest], arr[i]) # swap

heapify(arr, n, largest)

def sort\_heap(massiv\_to\_sort):

n = len(massiv\_to\_sort)

for i in range(n // 2, -1, -1):

heapify(massiv\_to\_sort, n, i)

for i in range(n - 1, 0, -1):

(massiv\_to\_sort[i], massiv\_to\_sort[0]) = (

massiv\_to\_sort[0],

massiv\_to\_sort[i],

) # swap

heapify(massiv\_to\_sort, i, 0)

def sort\_counting(input\_array):

M = max(input\_array)

count\_array = [0] \* (M + 1)

for num in input\_array:

count\_array[num] += 1

for i in range(1, M + 1):

count\_array[i] += count\_array[i - 1]

output\_array = [0] \* len(input\_array)

for i in range(len(input\_array) - 1, -1, -1):

output\_array[count\_array[input\_array[i]] - 1] = input\_array[i]

count\_array[input\_array[i]] -= 1

return output\_array

def sort\_bucket(massiv\_to\_sort):

max\_value = max(massiv\_to\_sort)

size = max\_value / len(massiv\_to\_sort)

buckets\_list = []

for x in range(len(massiv\_to\_sort)):

buckets\_list.append([])

for i in range(len(massiv\_to\_sort)):

j = int(massiv\_to\_sort[i] / size)

if j != len(massiv\_to\_sort):

buckets\_list[j].append(massiv\_to\_sort[i])

else:

buckets\_list[len(massiv\_to\_sort) - 1].append(massiv\_to\_sort[i])

for z in range(len(massiv\_to\_sort)):

sort\_insertion(buckets\_list[z])

final\_output = []

for x in range(len(massiv\_to\_sort)):

final\_output = final\_output + buckets\_list[x]

return final\_output

def countingSortForRadix(inputArray, placeValue):

countArray = [0] \* 10

inputSize = len(inputArray)

# equal to 2

for i in range(inputSize):

placeElement = (inputArray[i] // placeValue) % 10

countArray[placeElement] += 1

for i in range(1, 10):

countArray[i] += countArray[i - 1]

outputArray = [0] \* inputSize

i = inputSize - 1

while i >= 0:

currentEl = inputArray[i]

placeElement = (inputArray[i] // placeValue) % 10

countArray[placeElement] -= 1

newPosition = countArray[placeElement]

outputArray[newPosition] = currentEl

i -= 1

return outputArray

def sort\_radix(massiv\_to\_sort):

maxEl = max(massiv\_to\_sort)

D = 1

while maxEl > 0:

maxEl /= 10

D += 1

placeVal = 1

outputArray = massiv\_to\_sort

while D > 0:

outputArray = countingSortForRadix(outputArray, placeVal)

placeVal \*= 10

D -= 1

return outputArray

print("Генерирую тестовый массив")

test\_massiv = generate\_list(10000)

while True:

print(

"Как будем сортировать?\n1 - Bubble sort\n2 - Selection sort\n3 - Insertion sort\n4 - Merge sort\n5 - Quick sort\n6 - Timsort\n7 - Heap sort\n8 - Counting sort\n9 - Bucket Sort\n10 - Radix Sort\n"

)

selection = input()

if is\_number(selection):

selection = int(selection)

if selection == 1: # Сортировка пузырьком (Bubble Sort)

time\_start = time.perf\_counter()

test\_sorted = sort\_bubble(test\_massiv)

time\_finish = time.perf\_counter()

print(test\_sorted)

print("Время работы: " + str(time\_finish - time\_start))

elif selection == 2: # Сортировка выбором (Selection Sort)

time\_start = time.perf\_counter()

test\_sorted = sort\_selection(test\_massiv)

time\_finish = time.perf\_counter()

print(test\_sorted)

print("Время работы: " + str(time\_finish - time\_start))

elif selection == 3: # Сортировка вставками (Insertion Sort)

time\_start = time.perf\_counter()

test\_sorted = sort\_insertion(test\_massiv)

time\_finish = time.perf\_counter()

print(test\_sorted)

print("Время работы: " + str(time\_finish - time\_start))

elif selection == 4: # Сортировка слиянием (Merge Sort)

time\_start = time.perf\_counter()

test\_massiv\_2 = test\_massiv.copy()

sort\_merge(test\_massiv\_2)

time\_finish = time.perf\_counter()

print(test\_massiv\_2)

print("Время работы: " + str(time\_finish - time\_start))

elif selection == 5: # Быстрая сортировка (Quick Sort)

test\_massiv\_2 = test\_massiv.copy()

time\_start = time.perf\_counter()

test\_sorted = sort\_quick(test\_massiv\_2)

time\_finish = time.perf\_counter()

print(test\_sorted)

print("Время работы: " + str(time\_finish - time\_start))

elif selection == 6: # Тим-сортировка (Timsort)

time\_start = time.perf\_counter()

test\_sorted = sort\_timsort(test\_massiv)

time\_finish = time.perf\_counter()

print(test\_sorted)

print("Время работы: " + str(time\_finish - time\_start))

elif selection == 7: # Сортировка кучей (Heap Sort):

test\_massiv\_2 = test\_massiv.copy()

time\_start = time.perf\_counter()

sort\_heap(test\_massiv\_2)

time\_finish = time.perf\_counter()

print(test\_massiv\_2)

print("Время работы: " + str(time\_finish - time\_start))

elif selection == 8: # Сортировка подсчетом (Counting Sort)

time\_start = time.perf\_counter()

test\_sorted = sort\_counting(test\_massiv)

time\_finish = time.perf\_counter()

print(test\_sorted)

print("Время работы: " + str(time\_finish - time\_start))

elif selection == 9: # Блочная сортировка (Bucket Sort)

time\_start = time.perf\_counter()

test\_sorted = sort\_bucket(test\_massiv)

time\_finish = time.perf\_counter()

print(test\_sorted)

print("Время работы: " + str(time\_finish - time\_start))

elif selection == 10: # Сортировка радиксом (Radix Sort)

time\_start = time.perf\_counter()

test\_sorted = sort\_radix(test\_massiv)

time\_finish = time.perf\_counter()

print(test\_sorted)

print("Время работы: " + str(time\_finish - time\_start))

else:

print("Неверный ввод")

На рисунках 1 – 10 представлено время выполнения сортировки:



Рис. 1 – Bubble Sort



Рис. 2 – Selection Sort



Рис. 3 – Insertion Sort



Рис. 4 – Merge Sort



Рис. 5 – Quick Sort



Рис. 6 – Timsort



Рис. 7 – Heap Sort



Рис. 8 – Counting Sort



Рис. 9 – Bucket Sort



Рис. 10 – Radix Sor

На рисунке 11 представлено меню выбора для пользователя:

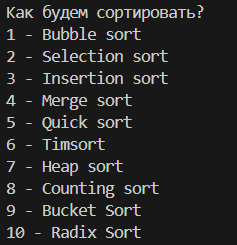


Рис. 11 – меню выбора сортировки

На рисунке 12 представлен пример вывода работы сортировки:

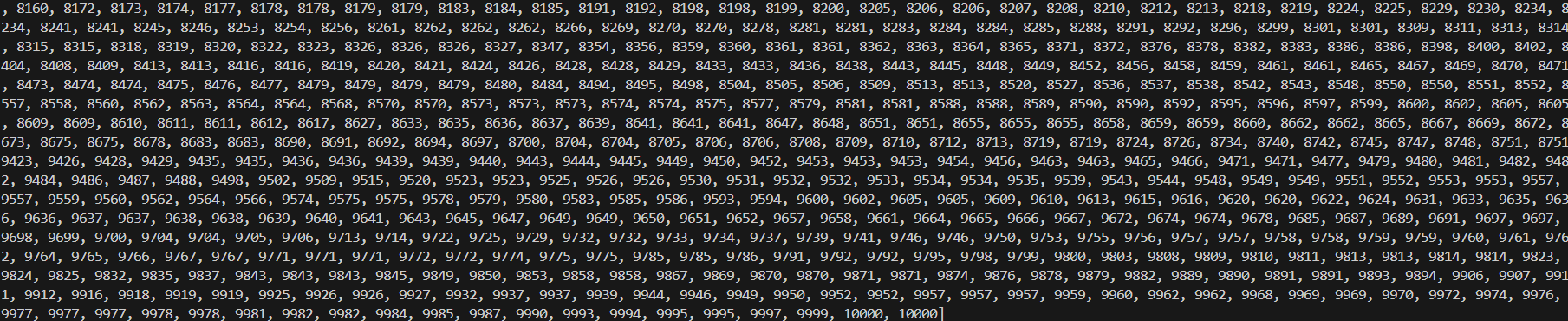


Рис. 12 – Вывод сортировки

**Вывод**

В ходе выполнения лабораторной работы были исследованы эффективности различных методов сортировки, используя язык программирования Python. Анализ эффективности каждой из сортировок был произведен в ходе выполнения следующих шагов:

1. Были написаны функции для различных методов сортировки (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Timsort, Heap Sort, Counting Sort, Bucket Sort, Radix Sort);
2. Произвели тест каждой из сортировок и записали время выполнения кода;
3. Подвели итоги по сортировкам, исходя из полученных данных.

Исследовав различные сортировки, по итогам лабораторной работы можно сделать вывод, что наиболее эффективной сортировкой для входных данных в виде массива из чисел является timsort.