МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Звіт
до лабораторної роботи з дисципліни
«Теорія алгоритмів та структур даних»

Виконав: Пфайфер В.В.
Група: ТР - 35
Прийняв: Андрущак В.С.

Львів 2021

Мета роботи: дослідження алгоритмів в середовищі Jupyter

Хід роботи:

1. Вибрати/знайти/згенерувати масив даних на 100 000 елементів.
2. Створити проект в середовищі Jupyter
3. Відсортувати масив даних згідно алгоритму сортування комірками
4. Побудувати графіки, який відображає наступне: - Теоретичний O(n) для часу (найкращий, середній і найгірший сценарій) - На цьому ж графіку побудувати O(n) для згенерованого масиву (крок вибрати самостійно) - Такий де графік побудувати для O(n) для використовуваної пам’яті
5. Здійснити удосконалення алгоритму сортування згідно заданого варіанту 1-3 способами.

```
#Bucket Sort Implementation in Python
https://stackabuse.com/bucket-sort-in-python/
def in_sort(array, left=0, right=None):
    if right is None:
        right = len(array) - 1

    # Loop from the element indicated by
    # `left` until the element indicated by `right`
    for i in range(left + 1, right + 1):
        # This is the element we want to position in its
        # correct place
        key_item = array[i]

        # Initialize the variable that will be used to
        # find the correct position of the element referenced
        # by `key_item`
        j = i - 1

        # Run through the list of items (the left
        # portion of the array) and find the correct position
        # of the element referenced by `key_item`. Do this only
        # if the `key_item` is smaller than its adjacent values.
        while j >= left and array[j] > key_item:
            # Shift the value one position to the left
            # and reposition `j` to point to the next element
            # (from right to left)
            array[j + 1] = array[j]
            j -= 1

        # When you finish shifting the elements, position
        # the `key_item` in its correct location
        array[j + 1] = key_item

    return array

def insertion_sort(array):
    min_run = 32
    n = len(array)
```

```
    # Start by slicing and sorting small portions of the
    # input array. The size of these slices is defined by
    # your `min_run` size.
    for i in range(0, n, min_run):
        in_sort(array, i, min((i + min_run - 1), n - 1))

    # Now you can start merging the sorted slices.
    # Start from `min_run`, doubling the size on
    # each iteration until you surpass the length of
    # the array.
    size = min_run
    while size < n:
        # Determine the arrays that will
        # be merged together
        for start in range(0, n, size * 2):
            # Compute the `midpoint` (where the first array ends
            # and the second starts) and the `endpoint` (where
            # the second array ends)
            midpoint = start + size - 1
            end = min((start + size * 2 - 1), (n-1))

            # Merge the two subarrays.
            # The `left` array should go from `start` to
            # `midpoint + 1`, while the `right` array should
            # go from `midpoint + 1` to `end + 1`.
            merged_array = merge(
                left=array[start:midpoint + 1],
                right=array[midpoint + 1:end + 1])

            # Finally, put the merged array back into
            # your array
            array[start:start + len(merged_array)] = merged_array

        # Each iteration should double the size of your arrays
        size *= 2


def bucket_sort(input_list):
    # Find maximum value in the list and use length of the list to determine which value in the list goes into which
bucket
    max_value = max(input_list)
    size = max_value/len(input_list)

    # Create n empty buckets where n is equal to the length of the input list
    buckets_list= []
    for x in range(len(input_list)):
        buckets_list.append([])

    # Put list elements into different buckets based on the size
    for i in range(len(input_list)):
        j = int (input_list[i] / size)
        if j != len (input_list):
            buckets_list[j].append(input_list[i])
        else:
```

```
            buckets_list[len(input_list) - 1].append(input_list[i])

    # Sort elements within the buckets using Insertion Sort
    for z in range(len(input_list)):
        insertion_sort(buckets_list[z])

    # Concatenate buckets with sorted elements into a single list
    final_output = []
    for x in range(len (input_list)):
        final_output = final_output + buckets_list[x]
    return final_output


def main():
    input_list = x
    print('ORIGINAL LIST:')
    #print(input_list)
    t1_start = process_time()
    sorted_list = bucket_sort(input_list)
    #print('sorted LIST:')
    #print(sorted_list)
    t1_stop = process_time()
    print("Elapsed time during the whole program in seconds:", t1_stop-t1_start)
```

Табл.1 Дослідження складності O(n) Сортування Комірками

Elements	50	100	500	1000	5000	10000	50000	100000
Worst case time, s	0	0	0	0.016	0.093	0.3281	11.1	121.7
Average case time, s	0	0	0	0.016	0.093	0.3125	8.4	90.1
Best case time, s	0	0	0	0.016	0.078	0.2812	8	79.8
Space, mB	83.7	83.8	83.9	84.1	84.8	85	88.5	88.6

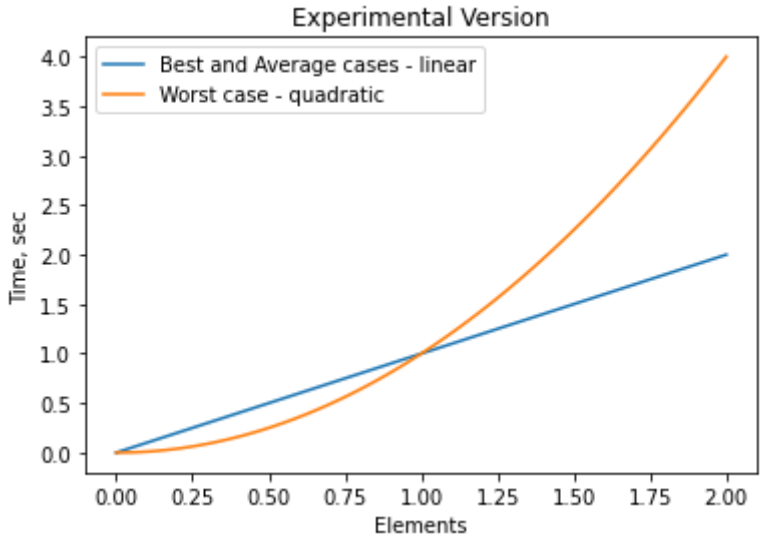

Рис.1 Експериментальна складність алгоритму

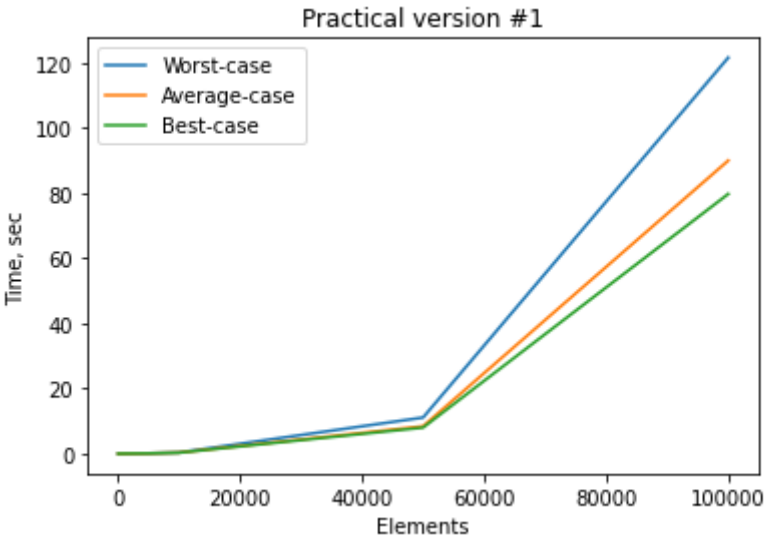

Рис.2 Практична складність алгоритму

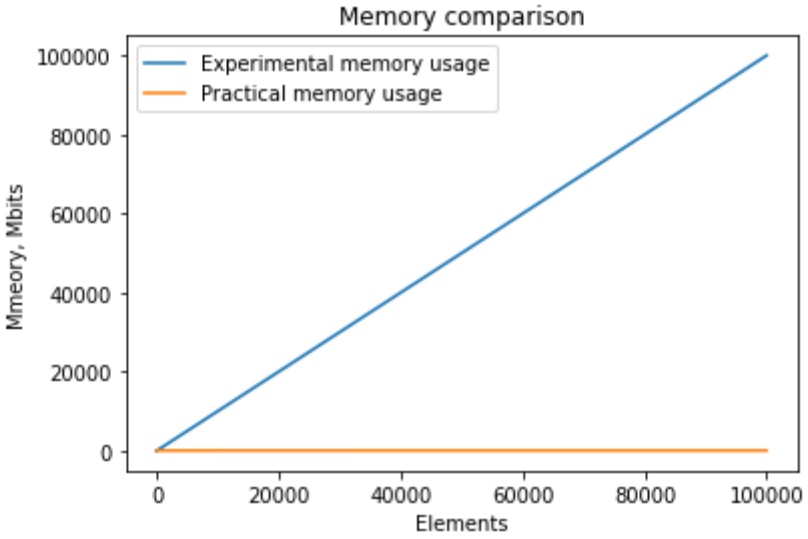

Рис.3 Порівняння використаної пам’яті

Let’s try to Improve the worst-case of Bucket-algorithm by changing the implemented algorithm of sorting in each bucket inside:

Elements	50	100	500	1000	5000	10000	50000	100000
Bubble. time, s	0	0	0	0.016	0.078	0.29687	10.31	101.33
Timsort. time, s	0	0	0	0.016	0.0625	0.29687	10.2	100

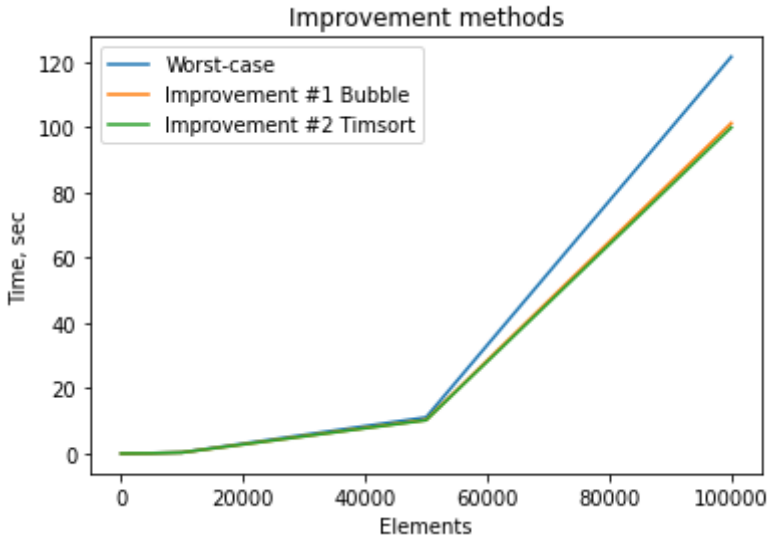

Рис.4 Дослідження покращень швидкодії алгоритму

Методи покращення - заміна сортування в кожному букеті з insertion sort на Bubble sort (1 випадок), Timsort (2 випадок)
Improvement Worst-case #1 Bubble Sort
https://realpython.com/sorting-algorithms-python/
Improvement Worst-case #2 Timsort
https://realpython.com/sorting-algorithms-python/

Висновок:
На даній лабораторній роботі досліджено алгоритм сортування комірками. Теоретично його складність в найкращому та середньому випадку - O(n), в найгіршому - O(n^2). Практично - квазілінійна функція, для всіх трьох випадків (масив з кількістю елементів до 100 000). Використання пам’яті: експериментально - O(n), практично - O(1). Проведено покращення найгіршого випадку даного алгоритму методом заміни алгоритму сортування всередині кожного букету.

Найгірша складність Якщо колекція, з якою ми працюємо, має невеликий діапазон (наприклад, той, який ми мали в нашому прикладі) - загальноприйнятим є багато елементів в одному сегменті, де багато сегментів порожні. Якщо всі елементи потрапляють в один сегмент, складність залежить виключно від алгоритму, який ми використовуємо для сортування вмісту самого сегмента. Оскільки ми використовуємо Insertion Sort - його найгірша складність сяє, коли список знаходиться в зворотному порядку. Таким чином, найгіршою складністю для сортування сегментів є також O (n2).

Складність найкращого випадку Найкращим випадком було б наявність усіх елементів, які вже відсортовані. Крім того, елементи розподілені рівномірно. Це означає, що кожен сегмент мав би однакову кількість елементів. З огляду на це, створення сегментів забирає O (n), а сортування вставки - O (k), надаючи нам складності O (n k). Складність середнього випадку

Середній випадок зустрічається у переважній більшості реальних колекцій. Коли колекція, яку ми хочемо сортувати, є випадковою. У цьому випадку для сортування відра для завершення потрібно O (n), що робить його дуже ефективним.