Sprawozdanie

Autorzy: Angelika Ostrowska, Kacper Straszak

Data: 24.03.2023

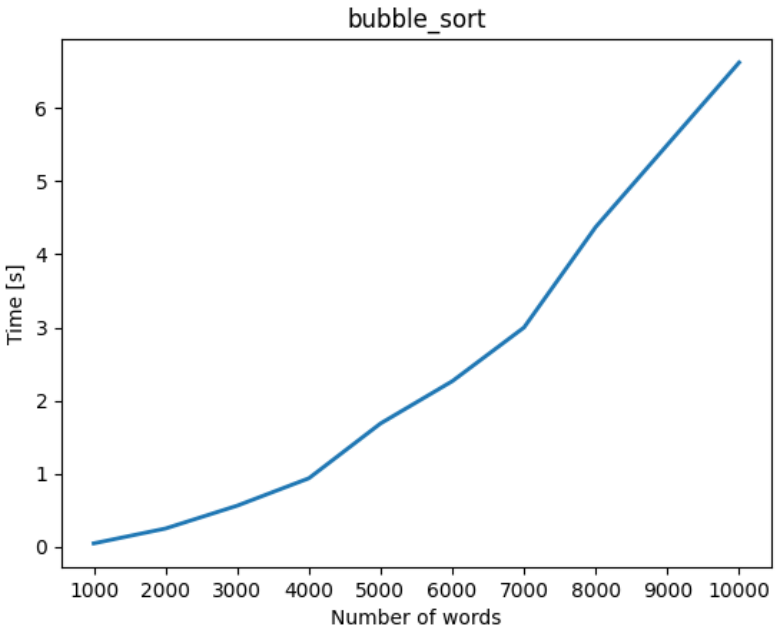
AISDI Zad2

Sortowanie bąbelkowe

```
def bubble_sort(list_to_sort):
    length = len(list_to_sort)
    for i in range(length-1):
        for j in range(length-i-1):
            if list_to_sort[j] > list_to_sort[j+1]:
                list_to_sort[j], list_to_sort[j+1] = list_to_sort[j+1],
list_to_sort[j]
    return list_to_sort
```

Bubble sort przyjmuje listę elementów. Algorytm porównuje dwa sąsiednie elementy i zamienia je miejscami, jeśli są w złej kolejności. Skutkiem każdej iteracji zewnętrznej pętli for jest umiejscowienie kolejnego największego elementu na odpowiedniej pozycji.

Metoda sortowania bąbelkowego ma średnią złożoność czasową O(n^2). Nie jest to wydajny algorytm, przy większych zbiorach czas sortowania może okazać się zdecydowanie za długi do wykorzystania w programie/aplikacji.

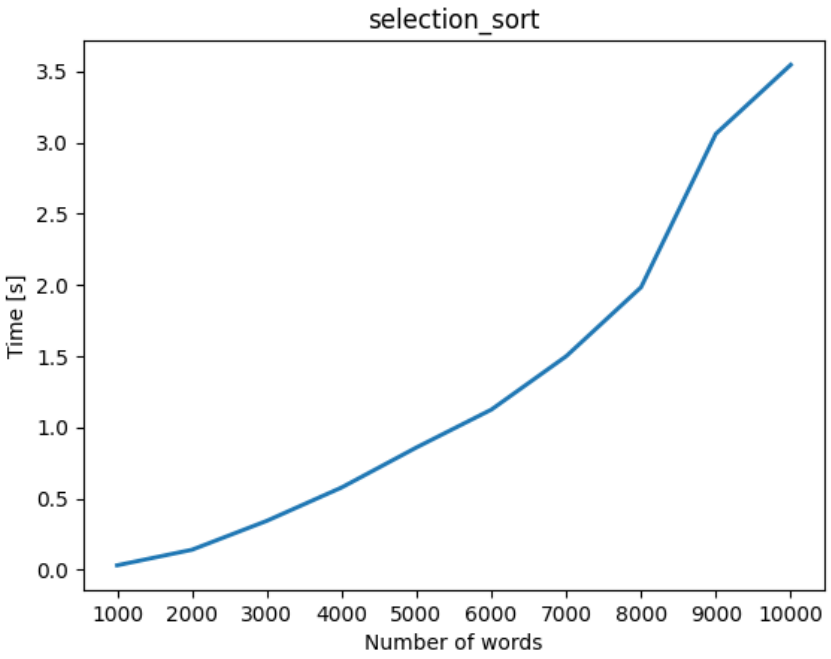


Sortowanie przez wybieranie

```
def selection_sort(list_to_sort):
    length = len(list_to_sort)
    for i in range(length-1):
        min_el = (list_to_sort[i], i)
        for j in range(i, length):
            if list_to_sort[j] < min_el[0]:
                min_el = (list_to_sort[j], j)
        list_to_sort[i], list_to_sort[min_el[1]] =
list_to_sort[min_el[1]], list_to_sort[i]
    return list_to_sort
```

Algorytm sortowania przez wybieranie polega na przechodzeniu przez nieposortowane jeszcze elementy, szukaniu najmniejszego wśród nich i umieszczaniu go na odpowiedniej pozycji w liście.
Szukanie najmniejszego elementu odbywa się poprzez porównywanie przeglądanego elementu z aktualnie najmniejszym elementem. Jeśli znaleziono nowy najmniejszy element, zapisujemy jego wartość i adres.
Znaleziony w danej iteracji i-ty najmniejszy element w liście zamieniamy z elementem o i-tym adresie. Kolejne najmniejsze elementy zostają ustawione na odpowiednich pozycjach.

Metoda sortowania przez wybieranie ma średnią złożoność czasową O(n^2). Jest to algorytm bardziej wydajny (przy odpowiednich wartościach) niż sortowanie bąbelkowe, lecz przy większej ilości elementów do posortowania wykonanie czynności może również zabrać dużą ilość czasu.

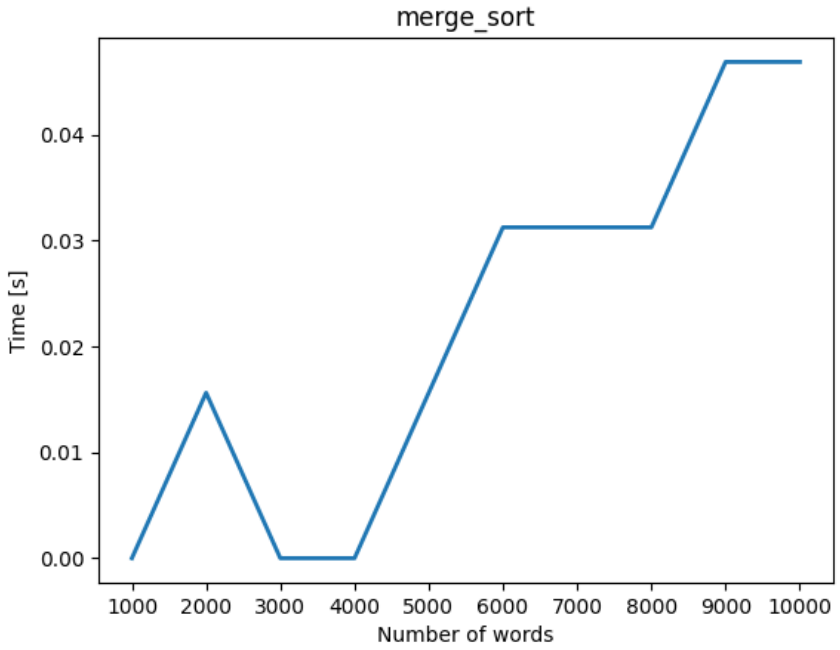


Sortowanie przez scalanie

```
def merge_sort(list_to_sort):
    return divide(list_to_sort)

def divide(list_to_sort):
    length = len(list_to_sort)
    if length == 1 or length == 0:
        return list_to_sort
    middle = ceil(length/2)
    left = divide(list_to_sort[:middle])
    right = divide(list_to_sort[middle:])
    return merge(left, right)

def merge(left, right):
    merged = []
    j_pointer = 0
    for i in range(len(left)):
        for j in range(j_pointer, len(right)):
            if left[i] < right[j]:
                merged.append(left[i])
                break
            else:
                merged.append(right[j])
                j_pointer = j + 1
        if j_pointer == len(right):
            merged += left[i:]
            break
    merged += right[j_pointer:]
    return merged
```

Algorytm sortowania przez scalanie składa się z dwóch głównych elementów - rekurencyjnego podziału list na połowy, a następnie łączeniu ich.
Samo sortowanie odbywa się na etapie scalania - funkcja mając dwie listy przegląda ich elementy, porównuje je i decyduje o ich pozycji w połączonej liście. Sklejamy coraz większe listy, aż w końcu otrzymamy posortowaną pierwotną listę.

Metoda sortowania przez scalanie ma średnią złożoność obliczeniową n log(n), dzięki rekurencji i dzieleniu listy na mniejsze fragmenty algorytm lepiej radzi sobie z dużą liczbą elementów (małe posortowane fragmenty łatwiej łączą się w duże części).



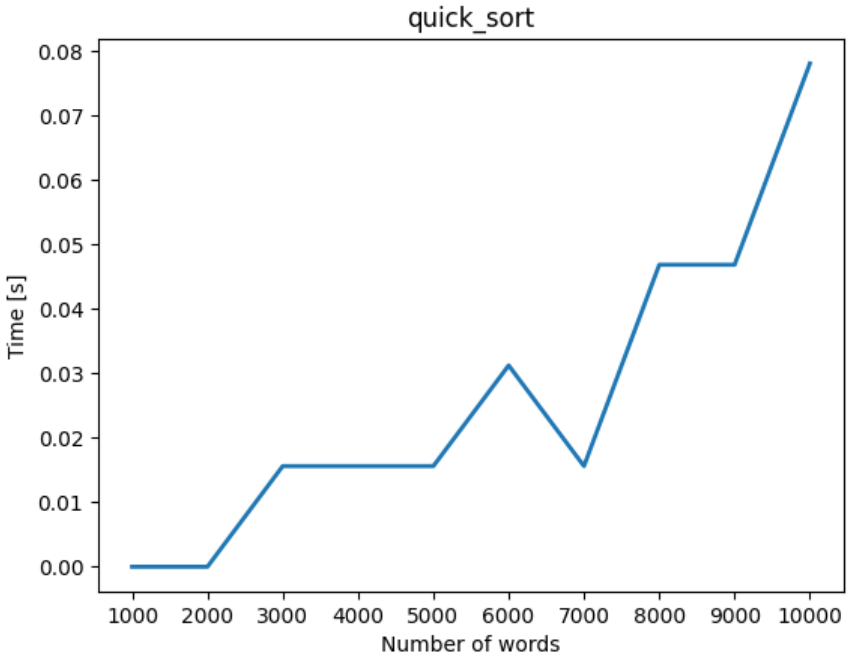
Sortowanie szybkie

```
def quick_sort(list_to_sort):

    length = len(list_to_sort)
    if length == 1 or length == 0:
        return list_to_sort

    pivot = -1
    bigger_left = 0
    smaller_right = length - 2

    found_bigger_left = False
    found_smaller_right = False

    for loop_number in range(length - 1):
        for i in range(bigger_left, length - 1):
            if list_to_sort[i] > list_to_sort[pivot]:
                bigger_left = i
                found_bigger_left = True
                break
        if not found_bigger_left:
            bigger_left = length-1
            break
```

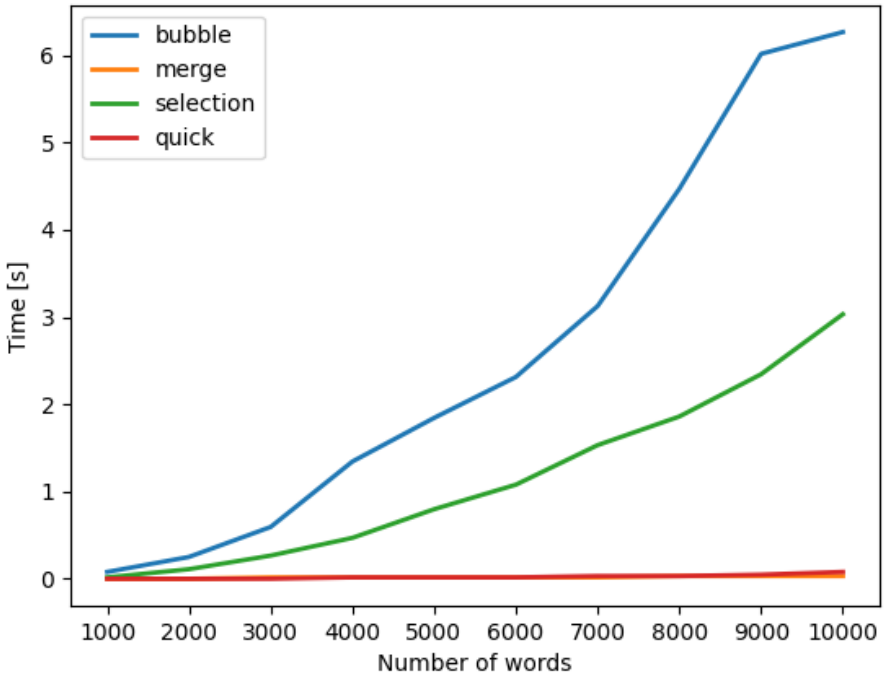
```
        for i in range(0, smaller_right + 1):
            if list_to_sort[smaller_right - i] < list_to_sort[pivot]:
                smaller_right = smaller_right - i
                found_smaller_right = True
                break
            if i == length - 2:
                list_to_sort[0], list_to_sort[pivot] =
list_to_sort[pivot], list_to_sort[0]
        if not found_smaller_right:
             bigger_left = 0
             break

        if bigger_left > smaller_right:
            list_to_sort[bigger_left], list_to_sort[pivot] =
list_to_sort[pivot], list_to_sort[bigger_left]
            break

        list_to_sort[bigger_left], list_to_sort[smaller_right] =
list_to_sort[smaller_right], list_to_sort[bigger_left]

    left_sorted = quick_sort(list_to_sort[:bigger_left])
    right_sorted = quick_sort(list_to_sort[bigger_left + 1:])
    pivot = [list_to_sort[bigger_left]]
    return left_sorted + pivot + right_sorted
```

Sortowanie szybkie również korzysta z rekursji. Algorytm na początku wybiera pivot (tutaj ostatni element aktualnej listy) i dzieli dane na dwie mniejsze listy, z elementami mniejszymi i większymi od pivota. Dla każdej z list zostaje ponownie wywołana funkcja sortowania i proces się powtarza. Łączymy posortowane już lewą i prawą stronę każdej z list, aż w końcu otrzymamy posortowaną listę wyjściową.

Metoda sortowania szybkiego ma średnią złożoność obliczeniową n log(n), dzięki rekurencji i dzieleniu listy na mniejsze fragmenty algorytm, tak samo jak sortowanie przez scalanie, dobrze radzi sobie z dużą liczbą elementów.



Porównanie algorytmów sortowania



Porównanie sortowania tego samego zbioru danych 4 różnymi algorytmami sortowania przedstawiliśmy na wykresie. Widać znaczącą różnicę między ich wydajnością, szczególnie przy większej liczbie elementów.
Wnioski: Najgorszym algorytmem okazało się sortowanie bąbelkowe, następnie sortowanie przez wybieranie. Lepiej wypadają algorytmy wykorzystujące zasadę “dziel i zwyciężaj” - sortowanie przez scalanie i sortowanie szybkie.