Zadanie 2 - sortowanie

Sformułowanie zadania

Zadanie polega na zaimplementowaniu dwóch algorytmów sortowania, odpowiednio po jednym z klasy O(n log n) i O(n) oraz określeniu ich złożoności czasowej i pamięciowej. Testowanymi algorytmami będą quicksort w wersji rekurencyjnej oraz radix sort.

Opis testowanych algorytmów

Quicksort

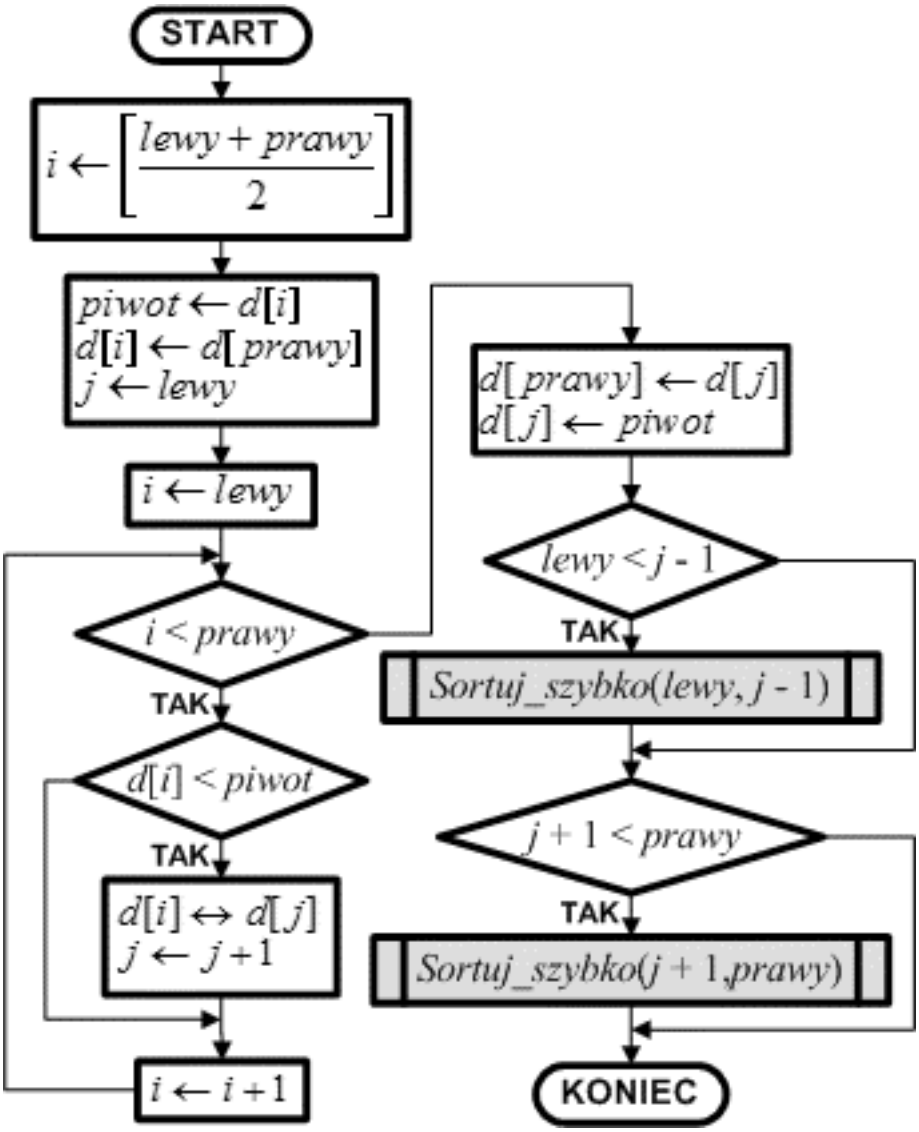
Quicksort (sortowanie szybkie) jest algorytmem sortowania w miejscu. Polega na rekursywnym wybieraniu spośród kluczy elementu rozdzielającego (pivota), a następnie dzielenie reszty kluczy na dwie pod-tablice, odpowiednio mniejsze i większe od pivota.

Pseudokod:

```
def quicksort(A, lo, hi):
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

def partition(A, lo, hi):
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Implementacja algorytmu opisana w reporcie jest rekurencyjna i zawsze wybiera pivot jako 1szy element tablicy.



Radix sort

Radix sort (sortowanie pozycyjne) jest rodzajem sortowania które polega na sortowaniu kluczy kolejno względem wartości ich kolejnych cyfr w danym systemie liczbowym. Sortujemy od najmniej znaczącej cyfry, do najbardziej znaczącej. Do kolejnych rund sortowania używamy najczęściej algorytmu counting sort (sortowanie przez zliczanie).

Algorytm ten jest bardzo popularny ponieważ jest czymś pomiędzy bucket sort a algorytmami sortowania in-place. Wadą bucket sort jest to, że musimy znać zakres wartości naszych danych aby dobrze określić ilość i zakres koszy do sortowania, co wpływa na zużycie pamięci algorytmu. Jednak dzięki parokrotnemu sortowaniu kolejnych cyfr, eliminujemy tą potrzebę, zakres wartości jest określony przez bazę wybranego systeu liczbowego. Ponadto, odpowiednio dobierając bazę, mamy bardzo granularną kontrolę nad ilością pamięci wykorzystywaną przez nasz algorytm, co

może bardzo przyśpieszyć sortowanie względem algorytmów o stałym zużyciu pamięci.

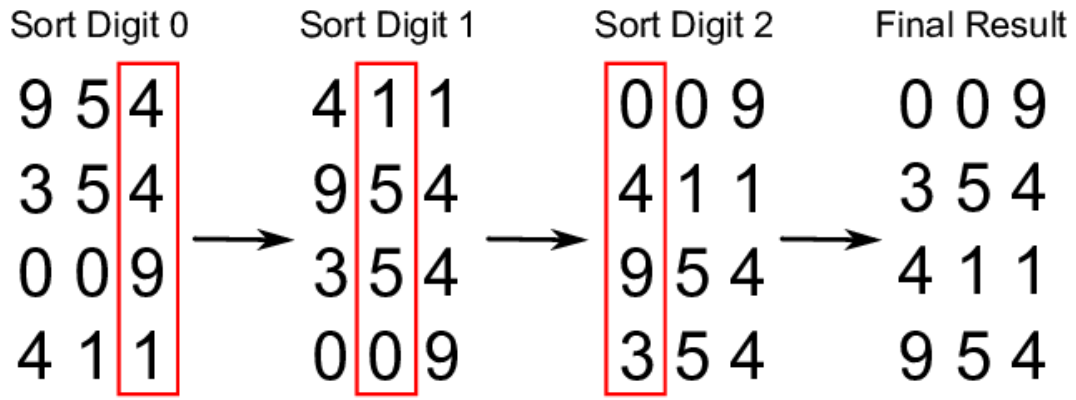
Pseudokod:

```
def countingsort(A, n):
    count = array of k+1 zeros
    for x in input do
        count[key(x)] += 1

    total = 0
    for i in 0, 1, ... k do
        count[i], total = total, count[i] + total

    output = array of the same length as input
    for x in input do
        output[count[key(x)]] = x
        count[key(x)] += 1

    return output
```



Implementacja algorytmu wykorzystuje operacje binarne i podstawy systemów liczbowych będące potęgami liczby 2 aby zapewnić należytą szybkość działania.

Rust stdsort

Domyślna stabilna funkcja sortująca z biblioteki standardowej języka Rust. Jej implementacja jest opisana następująco:

> Sorts the slice.
>
> This sort is stable (i.e., does not reorder equal elements) and O(n * log(n)) worst-case.
>
> When applicable, unstable sorting is preferred because it is generally faster than stable sorting and it doesn't allocate auxiliary memory. See sort_unstable.

Current implementation

The current algorithm is an adaptive, iterative merge sort inspired by timsort. It is designed to be very fast in cases where the slice is nearly sorted, or consists of two or more sorted sequences concatenated one after another.

Also, it allocates temporary storage half the size of self, but for short slices a non-allocating insertion sort is used instead.

Dane wejściowe i wyjściowe

Dane wejściowe generowane są przez generator liczb pseudolosowych zainicjalizowany stałym ziarnem, co zapewni niezmienność danych pomiędzy kolejnymi uruchomieniami programu. Po uruchomieniu programu i przetworzeniu pliku konfiguracyjnego, program kolejno będzie generował unikalne instancje względem rozmiaru (np. jeżeli w pliku konfiguracyjnym mamy dwie instancje o takim samym rozmiarze, to program wygeneruje jedną instancję o takim rozmiarze) do bufora read-only. Przy wykonywaniu przypadków testowych, wygenerowane instancje będą kopiowane do bufora roboczego na potrzeby wykonania sortowania.

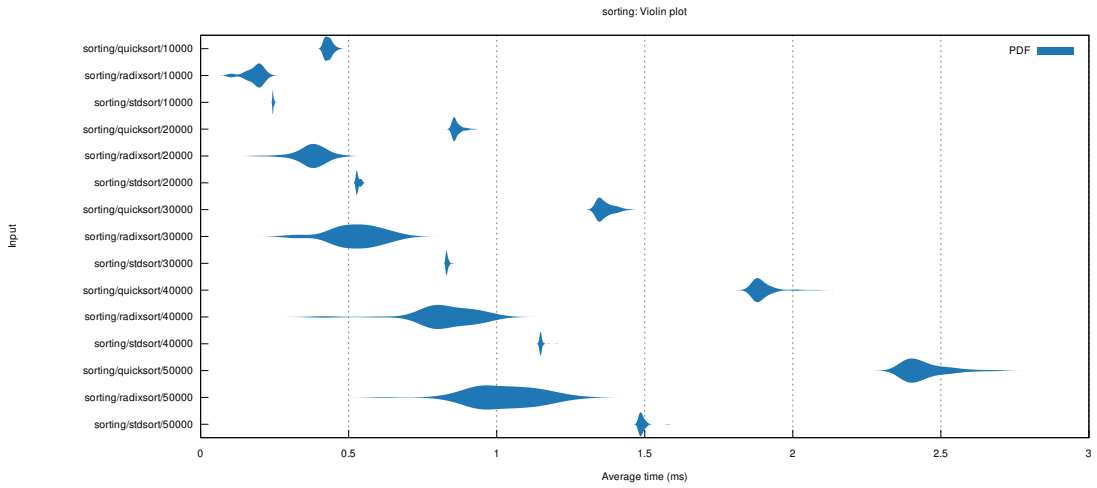
Procedura badawcza

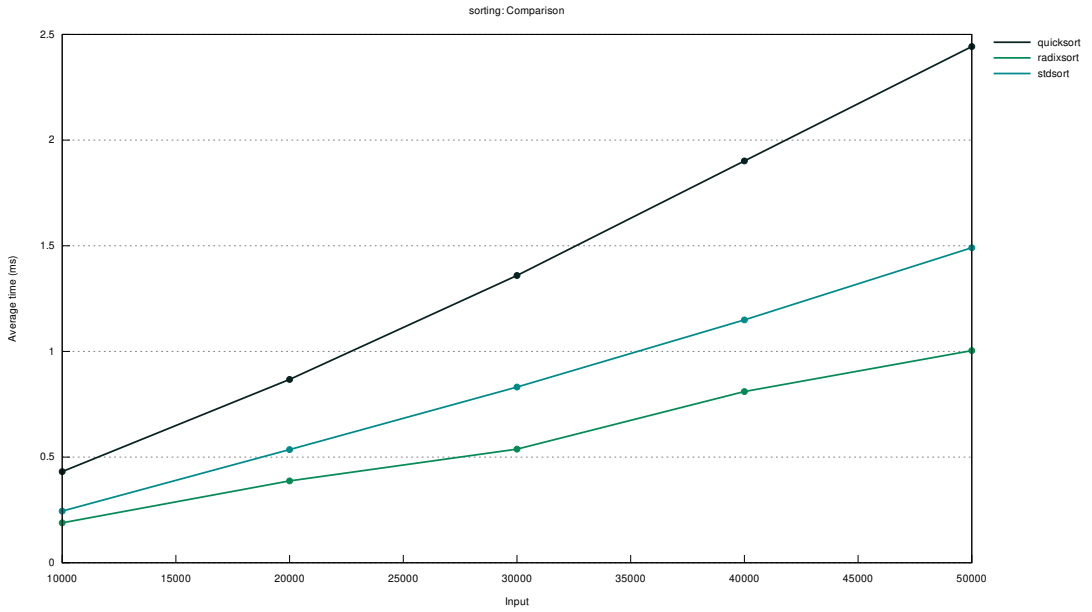
Do wykonania pomiarów wykorzystana została biblioteka Criterion-rs. Zapewnia ona narzędzia ułatwiające poprawne benchmarkowanie kodu i wylicza standardowe zmienne statystyczne, takie jak średnia czy odchylenie standardowe.

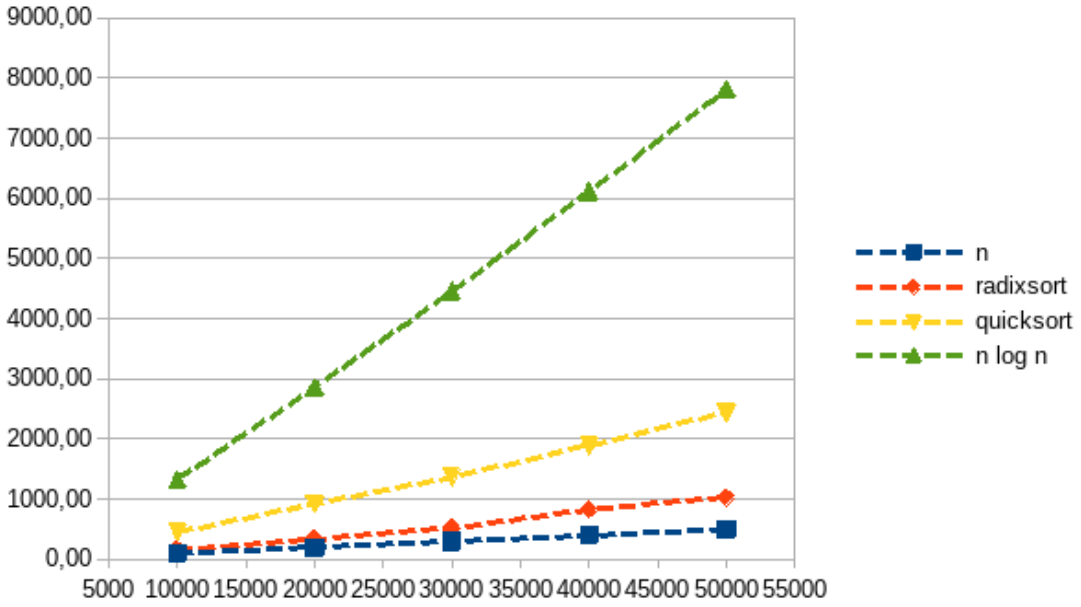
Proces pomiaru wygląda następująco:

- Rozgrzewka - Rutyna jest wykonywana wielokrotnie przez zadany czas by wypełnić i rozgrzać cache procesora i OS
- Pomiar - Rutyna jest wykonywana wielokrotnie i czasy wykonywania są mierzone i zapisywane
- Analiza - Mierzone próbki są analizowane i wyliczane są z nich zmienne statystyczne, takie jak średnia czy odchylenie standardowe

Wyniki







Wyniki pokazują że najwolniejszy jest quicksort, natomiast najszybszy jest radixsort.

Radixsort

Radixsort okazuje się być najszybszym z porównywanych algorytmów, jednak osiąga to kosztem znacznego zużycia pamięci.

W trakcie działania, radixsort wymaga utworzenia dwóch tablic:

1. Tablica o rozmiarze `N`, do której będziemy przenosić elementy z tablicy wejściowej

2. Tablica o rozmiarze `B - 1`, gdzie `B` to podstawa używanego systemu liczbowego

Podczas gdy tablica nr. 1 jest zależna od wielkości instancji problemu, rozmiar tablicy nr. 2 możemy kontrolować. Zwiększając podstawę systemu liczbowego, i w konsekwencji, rozmiar tablicy, jesteśmy w stanie wykorzystać spore pokłady pamięci obecne w nowoczesnych komputerach PC aby posortować wiele liczb relatywnie szybko.

Np. sortując po 16 bitów naraz, tj. przyjmując za podstawę systemu liczbowego `2^16 = 65 536` musimy utworzyć tablicę o długości 65 536 elementów. Zakładając że jest to tablica typu `int`, cała tablica zajmie wtedy `65 536 * 4B = 262 144B ~= 262kB`. Możemy wtedy sortować tablice typu `int` wykonując zaledwie dwie pętle, odpowiednio dla dolnych i górnych 16 bitów, których złożoność obliczeniowa wynosi O(n), tym samym złożoność całego algorytmu wynosi O(n).

Kiedy używać radixsorta:

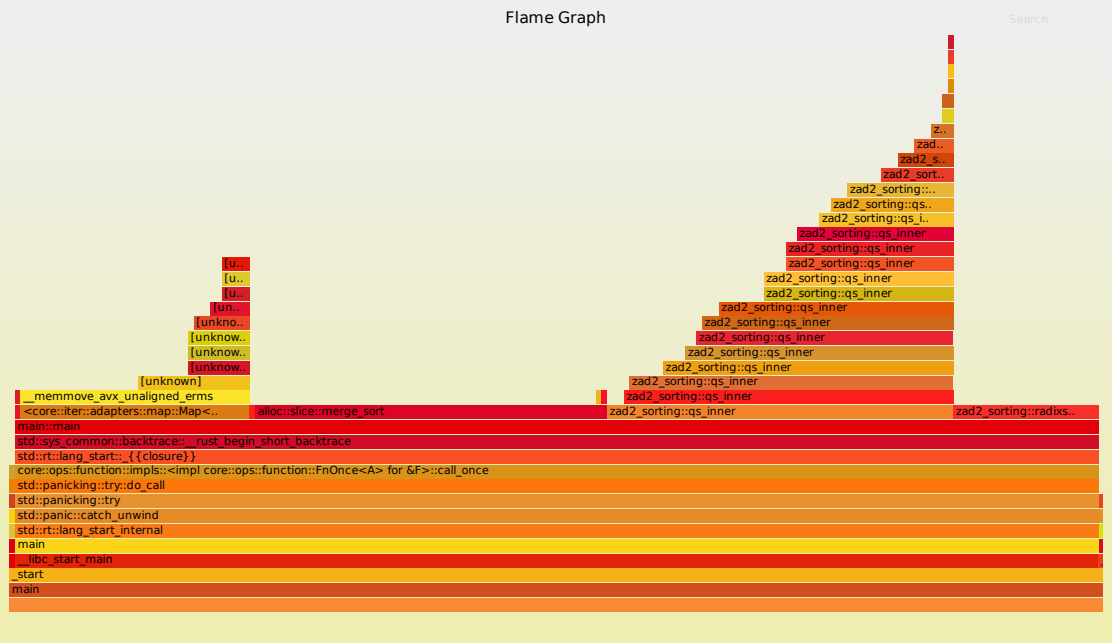
- Kiedy mamy dużo liczb do posortowania, i mamy dużo wolnej pamięci której możemy użyć

Kiedy *nie* używać radixsorta:

- Kiedy jest bardzo mało liczb do sortowania (wtedy czas potrzebny na setup, tj. alokację pamięci i spędzony na relatywnie większą ilość operacji wykonywanych dla każdego elementu, będzie większy niż czas wykonania jakiegoś innego algorytmu z klasy O(n log n), np. quicksort)
- Kiedy jesteśmy ograniczeni przez pamięć
- Kiedy na podstawie naszego elementu nie jesteśmy w stanie wyznaczyć klucza jako liczby całkowitej, tzn. nie możemy sortować np. liczb zmiennoprzecinkowych.

Quicksort

Quicksort okazuje się być najwolniejszym z porównywanych algorytmów, wolniejszy nawet od merge sorta z biblioteki standardowej. Dlaczego? By dowiedzieć się więcej, możemy użyć narzędzia flamegraph, prezentującego stos wywołania programu w różnych punktach jego wykonania:



Jak można zatem się domyślić, prawdopodobne jest iż rekurencyjna implementacja znacząco zwalnia nasz algorytm, szczególnie dla małych podtablic (ok. 10 elementów), gdzie ilość pracy niezbędna do samego wywołania funkcji jest większa niż ilość pracy poświęcona na sortowanie.