Analiza empiryczna złożoności algorytmów

Aleksander Ordyczyński, w65557

sformułowanie problemu

Celem projektu jest napisanie algorytmów wyszukiwania i zaprezentowanie ilości operacji elementarnych i dominujących w zależności od rozmiaru zadania oraz typu danych.

opis problemu

1. Problem wyszukiwania
  * Zadaniem jest zbiór elementów i element wyszukiwany.
  * Pytanie - czy element należy do zbioru.
  * Konieczna jest możliwość porównania elementów

2. Problem sortowania
  * zadaniem jest skończony ciąg.

3. Operacja elementarna to operacje podstawowe, możemy takie wyróżnić czytając kod w języku Assembly. Do operacji elementarnych należą:
  * operacje matematyczne: + - * / %
  * operacje logiczne: | & ~ || && !
  * operacje relacyjne: == != > >= < <=
  * przypisanie: zmienna = wartość
  * dostęp do elementu tablicy: tablica[index]
  * powrót na początek pętli

4. Operacje dominujące - liczba wykonań wszystkich operacji elementarnych może być trudna do wyznaczenia. Dlatego określamy operacje dominujące:
  są to takie operacje elementarne, że wykonaniu dowolnej operacji elementarnej towarzyszy wykonanie operacji dominującej.

3. opis rozwiązania

Kod programu dzieli się na następujące pliki:
* project.c - główny program, tu jest main
* project.h - plik nagłówkowy, tutaj są deklaracje wszystkich funkcji
* sort_search.h - algorytmy sortujące i wyszukiwania
* generation.h - funkcje generujące dane z algorytmów

Główny plik programu jest bardzo mały, wywołuje on jedynie funkcje generujące dane z algorytmów pomaga on sensownie połączyć wszystko w całość.

```c
#include "project.h"
#include "generation.h"
#include "sort_search.h"

int main()
{
    srand(time(NULL));

    // generuj wyniki testu do katalogu data
    run_sort_test(selection_sort, "selection_sort", TEST_SIZE);
    run_sort_test(insert_sort, "insert_sort", TEST_SIZE);
    run_sort_test(merge_sort, "merge_sort", TEST_SIZE);

    return 0;
}
```

plik: project.c

Plik nagłówkowy project.h może zostać zastosowany do prezentacji struktury programu, tu funkcje pomocnicze:

```c
// funkcje przeprowadzające testy
void run_sort_test(void (*sort)(int *, int), char *sort_name, int test_num);
void clear_files(char *sort_name);
void save_results(int n, char *sort_name, char *input_type);

// funkcje generujące tablice
int* rand_array(int n);
int* sorted_array_full(int n, bool inversed);
void rand_decrement_num(int *x);
void rand_increment_num(int *x);
```

plik: project.h

tutaj funkcje sortujące i wyszukiwania:

```c
// algorytmy sortujące i funkcje pomocnicze
void insert_sort(int *array, int n);

void selection_sort(int *array, int n);
void swap(int *x1, int *x2);

void merge_sort(int *a, int n);
int* copy_array(int* array, int n);
void mergesplit(int B[], int istart, int iend, int A[]);
void merge(int A[], int istart, int imiddle, int iend, int B[]);

// algorytmy wyszukiwania
int linear_search(int value, int *array, int n);
bool binary_search(int value, int *array, int n);
```

plik: project.h

Plik generation.h jest bardzo długi, zamierzam go tu opisać tylko częściowo resztę informacji można znaleźć w kodzie źródłowym.

Plik zaczyna się od deklaracji zmiennych globalnych o nazwach op_e i op_d będących licznikami operacji elementarnych i dominujących.

Następnie mamy główną funkcję projektu:

```c
// wykonaj test_num*3(dane losowe, posortowane i posortowane odwrotnie)
// funkcji sortującej (*sort), zapisz do pliku o nazwie
// nazwa_sortowania-typ_danych.txt w folderze data
void run_sort_test(void (*sort)(int *, int), char *sort_name, int test_num)
{
    clear_files(sort_name);  // wyczyść poprzednie wyniki programu
    int *array;
    for (int n = 1; n <= test_num; n++) {
        array = rand_array(n);
        (*sort)(array, n);  // merge_sort dla run_sort_test(merge_sort, ...
        save_results(n, sort_name, "random");
        free(array);

        array = sorted_array_full(n, false);
        (*sort)(array, n);
        save_results(n, sort_name, "sort");
        free(array);

        array = sorted_array_full(n, true);
        (*sort)(array, n);
        save_results(n, sort_name, "inversed_sort");
        free(array);
    }
}
```

Interesujący jest tutaj zapis
> void (*sort)(int *, int)
jest to argument będący pointerem do funkcji, dzięki niemu możemy potem aktywować funkcję dla każdego z algorytmów sortowania w main'ie.

Funkcja generuje jeden z 3 typów tablic: posortowaną, posortowaną odwrotnie i z elementami losowymi, na każdej z tych 3 tablic wykonuje sortowanie przekazane jako argument funkcji, następnie zapisuje operacje dominujące i elementarne dla rozmiaru n do pliku o nazwie
> *nazwa_algorytmu_sortowania-typ_danych_wejsciowych.txt*
w folderze dane i uwalnia miejsce w pamięci przeznaczone na alokację tablicy.

Wszystko to wykonuje się test_num razy.

Funkcja sorted_array_full generuje posortowaną lub posortowaną odwrotnie tablicę,
tak jak funkcja wyżej, ta funkja także korzysta z pointerów funkcji.
Tutaj, w zależności od wartości zmiennej inversed, modify_num będzie zmniejszał lub zwiększał wartość x o losową, niedużą liczbę.
Dzięki temu jesteśmy w stanie niskim kosztem wygenerować posortowaną tablicę, alternatywnym rozwiązaniem byłoby sortowanie tablicy wygenerowanej przez funkcję rand_array.
full w nazwie zostawia możliwość definicji macro sorted_array i inverted_sorted_array aktywującego array_sort z inversed = false lub inversed = true

```
// zwróć pointer do tablicy o rozmiarze n
// funkcja zwraca tablice posortowaną lub posortowaną odwrotnie
// w zależności od wartości zmiennej inversed
int* sorted_array_full(int n, bool inversed)
{
    int *array;
    array = malloc(n * sizeof (int));

    void (*modify_num)(int *) =
        (inversed) ? &rand_decrement_num : &rand_increment_num;

    int x = rand() % RAND_SIZE;    // wartość pierwszego elementu tablicy

    // zwiększa wartość pierwszego elementu,
    // dzięki temu liczby w tablicy nie będą miały wartości negatywnych
    if (inversed)
        x += n * RAND_SIZE_MODIFY;

    // wypełnij tablicę rosnącymi/malejącymi wartościami
    for (int i = 0; i < n; i++) {
        (*modify_num)(&x);  // aktywuje rand_increment_num
                            //  lub rand_decrement_num
        array[i] = x;
    }

    return array;
}
```

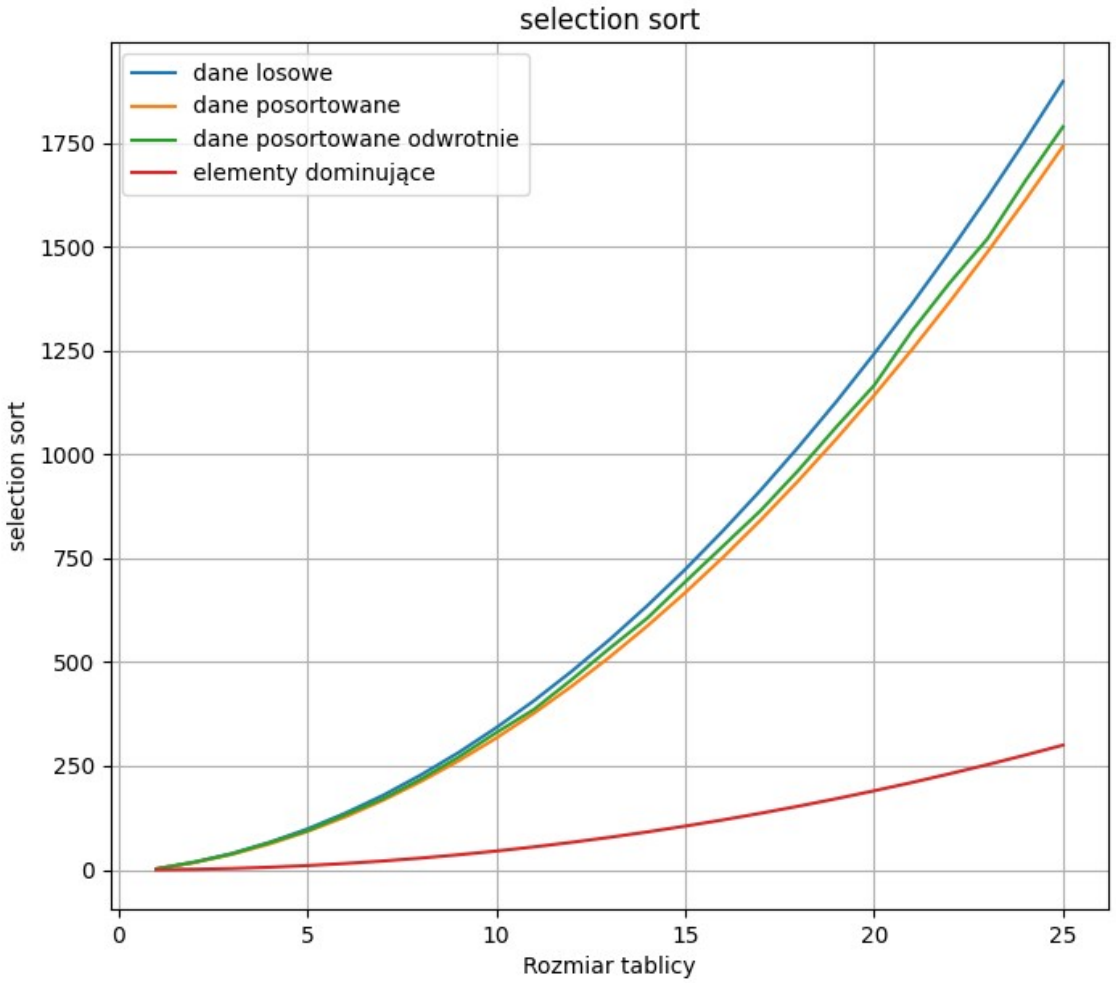
sorted_array_full działa dzięki poniższym funkcjom:

```c
// zwiększ wartość x o pseudolosową liczbę
// o maksymalnym rozmiarze stałej RAND_SIZE_MODIFY
void rand_increment_num(int *x)
{
    *x += rand() % RAND_SIZE_MODIFY + 1;
}

// zmniejsz wartość x o pseudolosową liczbę
// o maksymalnym rozmiarze stałej RAND_SIZE_MODIFY
void rand_decrement_num(int *x)
{
    *x -= rand() % RAND_SIZE_MODIFY + 1;
}
```

W pliku sort_search.h mamy wszystkie testowane algorytmy,
każdy z nich po wykonaniu operacji elementarnej inkrementuje wartość op_e
i wartość op_d po wykonaniu operacji dominującej.

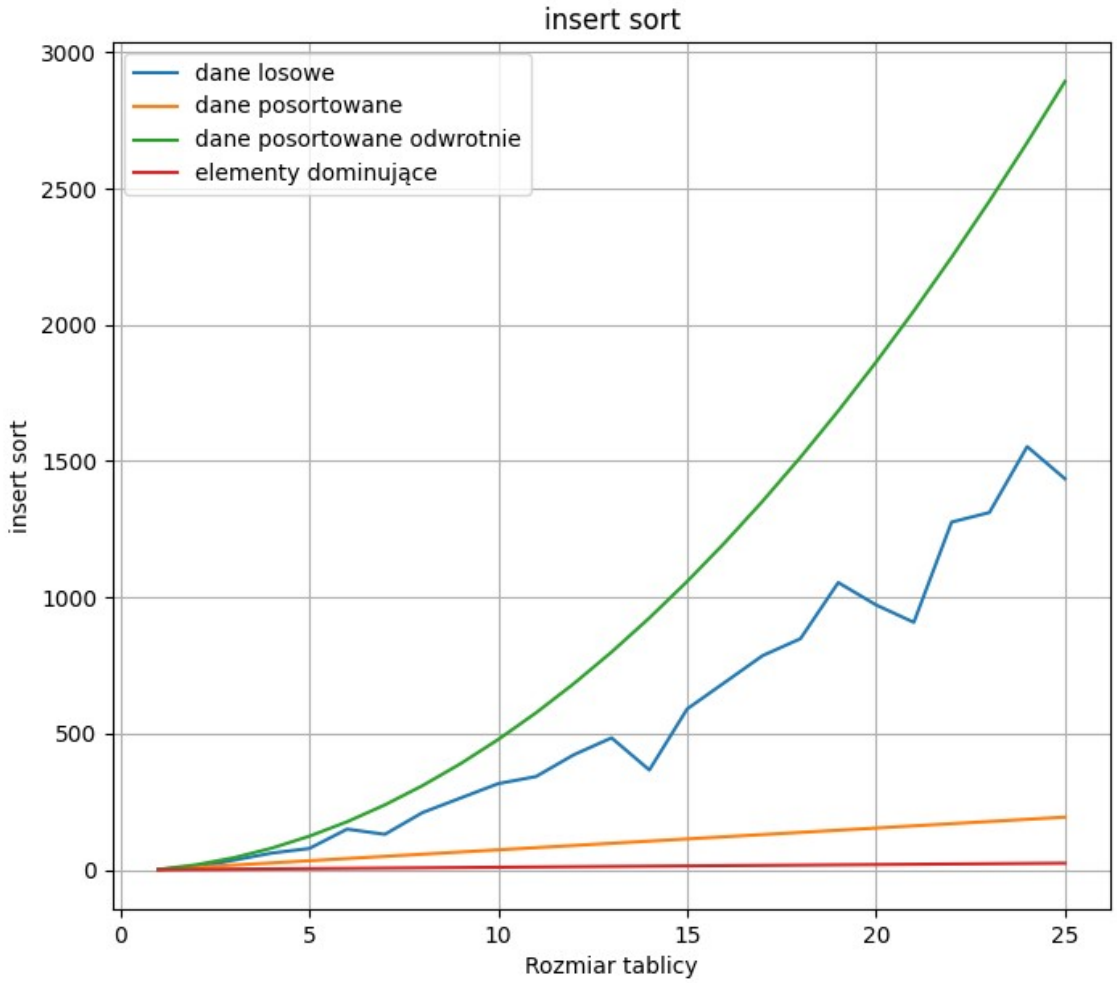
```c
// sortuj tablicę array o rozmiarze n
void insert_sort(int *array, int n)
{
    int temp, j, i;
    for (i = 1, op_e++; i < n; i++, op_e+=2) {
        temp = array[i]; op_e += 2;
        op_e+=2;
        op_d++;
        for (j = i-1; j >= 0 && temp < array[j]; j--, op_e+=5) {
            array[j+1] = array[j]; op_e+=4;
        }

        array[j+1] = temp; op_e+=2;
    }
}
```

Prezentacja wyników
Selection sort

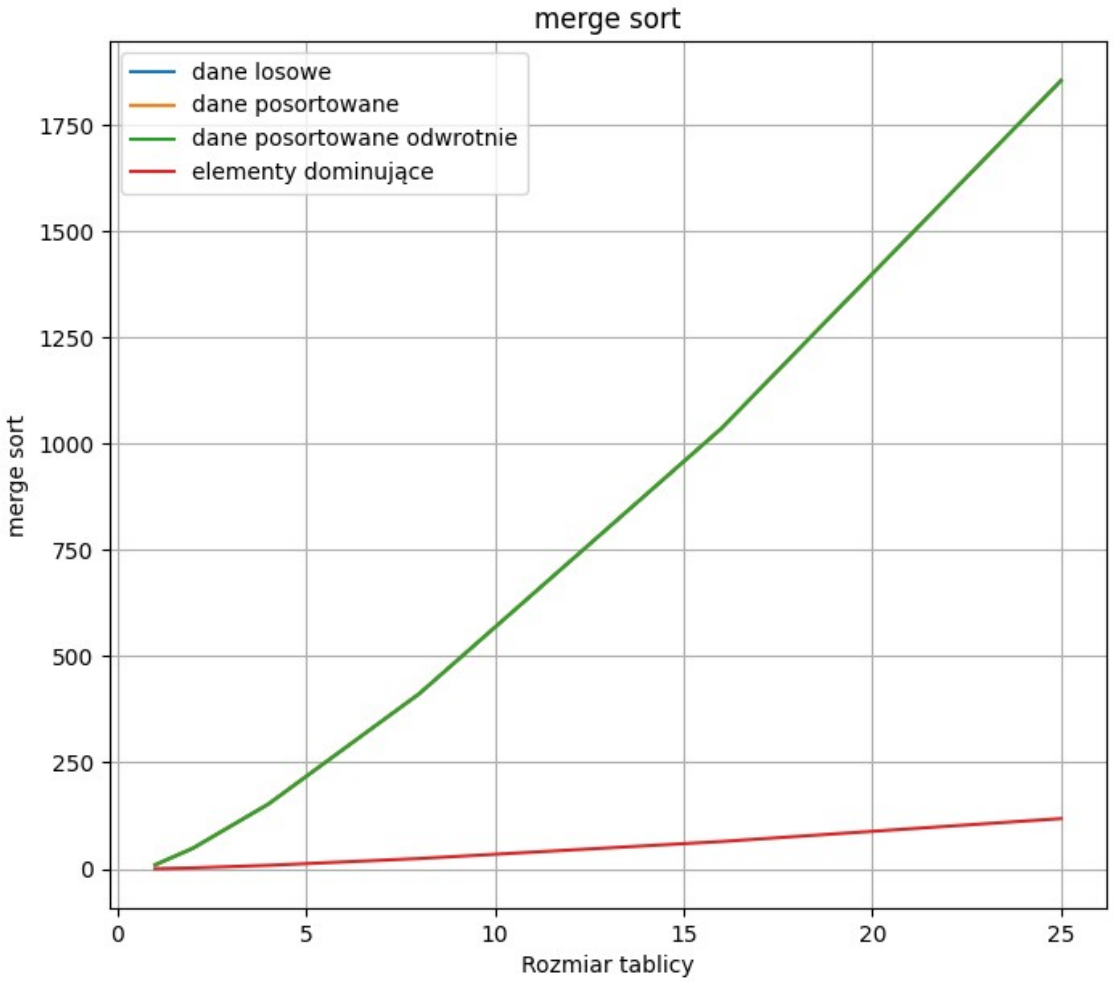


Liczba wartości

Insertion Sort



Merge Sort


merge sort
dane losowe
dane posortowane
dane posortowane odwrotnie
elementy dominujące
merge sort
1750
1500
1250
1000
750
500
250
0
0
5
10
15
20
25
Rozmiar tablicy

wnioski

Wykresy każdego z algorytmów sortowania znacząco się od siebie różnią, najwięcej wspólnego mają ze sobą merge sort i selection sort,w algorytmie sortowania przez merge liczba operacji elemntarnych nakłada się - są one wykonywane tylko w funkcji łączącej która wykonuje się niezależnie od typu zadania, z tego powodu ich liczba zawsze będzie taka sama.
Linie w wykresie algorytmu przez selekcje na siebie się nakładają.
Wykres algorytmu sortowania przez Insertion wygląda tak jak wyobrażałem sobie każdy wynik co mnie szczególnie zaskoczyło.

Autorem programu i projektu jest Aleksander Ordyczyński