

## Spis treści

1. Wstęp.....	3
2. Ogólny opis algorytmu rozwiązania.....	4
3. Pliki zewnętrzne i biblioteki.....	6
4. Zmienne globalne.....	6
5. Funkcje.....	7
6. Dokumentacja działania programu.....	10
7. Kod źródłowy.....	12

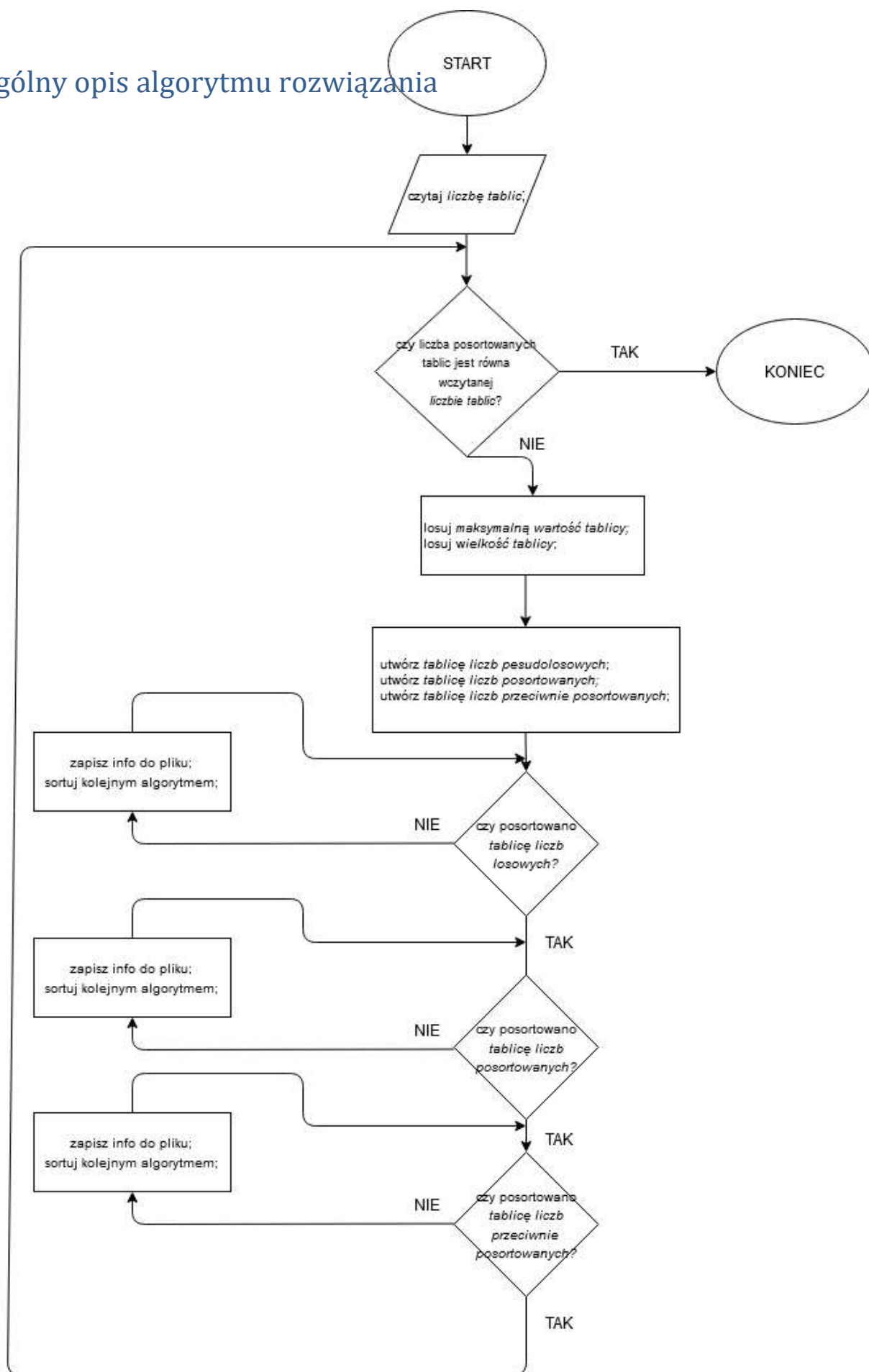
## 1. Wstęp

Celem programu jest otrzymanie danych dotyczących czasu wykonywania się funkcji sortujących tablicę danych o zmiennej długości, zmiennym dopuszczalnym zakresie wartości, i zapisanie otrzymanych danych w pliku tekstowym.

Program ma spełniać następujące wymagania:

- Zawierać implementację co najmniej kilku algorytmów sortujących, różniących się złożonością obliczeniową. Zaimplementowane mają być algorytmy o różnych rzędach złożoności (zgodnie z notacją „Dużego O”), w szczególności  $O(n^2)$  oraz  $O(n \cdot \log(n))$ .
- Sortowaniu mają być poddawane tablice liczb całkowitych nieujemnych.
- Program będzie sam generował tablicę liczb pseudo-losowych. Jej wielkość i maksymalna dopuszczalna wartość również będzie generowana pseudo-losowo, podlegając ograniczeniu zapisanemu w dyrektywie preprocesora.
- Użytkownik będzie wprowadzał liczbę pseudo-losowych tablic, poddawanych następnie procedurze sortowania. Każda tablica będzie sortowana wszystkimi zapisanymi w programie algorytmami.
- Sortowaniu będą również poddawane modyfikacje pseudo-losowej tablicy: tablica pochodna posortowana rosnąco oraz tablica pochodna posortowana malejąco.
- W celu obliczenia czasu wykonywania się danej instrukcji (wywołanie funkcji implementującej algorytm sortujący), wykorzystany powinien być czas procesora, podzielony przez stałą będącą liczbą jednostek czasu procesora na sekundę.
- Wynik w postaci: nazwy algorytmu, liczby elementów w tablicy, maksymalnej dopuszczalnej wartości w tablicy oraz czasu wykonywania operacji sortowania, powinien być zapisany w pliku tekstowym.

## 2. Ogólny opis algorytmu rozwiązania



Na poprzedniej stronie umieszczono uproszczony schemat blokowy funkcji głównej programu. Program wymaga wprowadzenia od użytkownika tylko jednej wielkości – **liczby tablic liczb pseudolosowych**, które mają być poddane pętli sortowania różnymi algorytmami (wraz z sortowaniem tablic pochodnych – posortowanej i przeciwnie posortowanej).

Program następnie generuje w odpowiedniej kolejności szereg wielkości pseudolosowych (wielkość tablic, poszczególne elementy tablic). Wskaźniki do tak utworzonych tablic przekazywane są następnie funkcjom, stanowiącym implementacje algorytmów sortujących, przy czym mierzony jest czas od wywołania funkcji do jej usunięcia ze stosu. Należy zaznaczyć, że samo sortowanie odbywa się na kopiach pierwotnych tablic, ponieważ algorytmy sortujące modyfikują tablicę wejściową.

Gdy zakończy się sortowanie każdą metodą, generowane są parametry tablic: maksymalna możliwa wartość elementu oraz wielkość tablicy. Te dwa argumenty przesyłane są do funkcji generującej pseudo-losową tablicę. Tworzone są 2 tablice pochodne: posortowana rosnąco i posortowana malejąco. Następnie wcześniej opisana procedura powtarza się. Pętla główna ulega terminacji, gdy wyczerpiemy wszystkie liczby naturalne mniejsze lub równe od wprowadzonej **liczby tablic liczb pseudolosowych**.

### 3. Pliki zewnętrzne i biblioteki

a) Wykorzystane pliki zewnętrzne:

**BRAK**

b) Wykorzystane biblioteki:

- <stdio.h>
  - W celu możliwości obsługi funkcji wejścia i wyjścia (do/z pliku, do/z standardowego wejścia/wyjścia)
- <stdlib.h>
  - W celu korzystania z funkcji do dynamicznego przydzielania i zwalniania pamięci.
- <time.h>
  - W celu generowania liczb pseudolosowych.
  - W celu wyznaczenia czasu sortowania.

### 4. Zmienne globalne

W programie nie wykorzystano żadnych zmiennych globalnych.

Wykorzystano jednak kilka dyrektyw preprocesora:

<code>#define MAX_NUM 2000000</code>	MAX_NUM to największa możliwa liczba, która może zostać wygenerowana jako wartość elementu tablicy
<code>#define MAX_SAMPLE_SIZE 100000</code>	MAX_SAMPLE_SIZE to największa możliwa wielkość tablicy liczb pseudolosowych
<code>#define MAX_CHAR 30</code>	MAX_CHAR to największa możliwa nazwa funkcji, która to nazwa jest elementem tablicy w strukturze
<code>#define SORT_ALG_N 5</code>	SORT_ALG_N to liczba implementacji algorytmów sortujących, które jako jeden z argumentów przyjmują wielkość tablicy
<code>#define SORT_ALG_ST 2</code>	SORT_ALG_ST to liczba implementacji algorytmów sortujących, które jako jeden z argumentów przyjmują początkowy i końcowy indeks sortowanej tablicy

## 5. Funkcje

Lp.	Nagłówek funkcji	Nazwa funkcji	Typ i znaczenie zwracanej wartości	Typy i znaczenie parametrów funkcji	Ogólny algorytm działania	Uwagi
1	void mergeSort(int *arr, int start, int finish)	mergeSort	void	int *arr: wskaźnik na tablicę liczb całkowitych int start: indeks pierwszego elementu tablicy int finish: indeks ostatniego elementu tablicy	Jest to rekurencyjna funkcja będąca implementacją algorytmu sortowania przez scalanie. Tablica, której wskaźnik jest argumentem funkcji, jest w niej modyfikowana tak, że efektem jest tablica posortowana rosnąco od start do finish.	Funkcja wywołuje inną funkcję w swoim ciele – merge.
2	void merge(int *arr, int start, int middle, int finish)	merge	void	int *arr: wskaźnik na tablicę liczb całkowitych, posortowanych od start do middle i od (middle+1) do finish. int start: indeks pierwszego elementu pierwszej tablicy int middle: indeks ostatniego elementu pierwszej części tablicy int finish: indeks ostatniego elementu drugiej części tablicy	Funkcja ta sortuje tablicę złożoną z 2 części – pierwszej posortowanej rosnąco (o indeksach od start do middle) i drugiej posortowanej rosnąco (od middle+1 do finish). Tablica, której wskaźnik jest argumentem funkcji, jest w niej modyfikowana tak, że efektem jest tablica posortowana rosnąco od start do finish.	Ta funkcja jest wywoływana wyłącznie przez funkcję mergeSort.
3	void printArray(int *arr, int n)	printArray	void	int *arr: wskaźnik na tablicę liczb całkowitych int n: liczba elementów tablicy	Funkcja drukuje na standardowe wyjście tablicę liczb całkowitych o długości n.	-
4	void bubbleSort(int *arr, int n)	bubbleSort	void	int *arr: wskaźnik na tablicę liczb całkowitych int n: liczba elementów tablicy	Jest to implementacja algorytmu sortowania bąbelkowego. Tablica, której wskaźnik jest argumentem funkcji, jest w niej modyfikowana tak, że efektem jest tablica posortowana rosnąco.	Funkcja wywołuje inną funkcję w swoim ciele – swap.
5	void swap (int *a, int *b)	swap	void	int *a: wskaźnik na zmienną typu całkowitego int *b: wskaźnik na zmienną typu całkowitego	Funkcja zamienia miejscami 2 zmienne. Wskaźnik na pierwszą zmienną wskazuje na drugą, a wskaźnik na drugą zmienną wskazuje na pierwszą. Argumenty są przekazane przez referencję.	-
6	void insertionSort(int *arr, int n)	insertionSort	void	int *arr: wskaźnik na tablicę liczb całkowitych	Jest to implementacja algorytmu sortowania przez wstawianie.	-

				int n: liczba elementów tablicy	Tablica, której wskaźnik jest argumentem funkcji, jest w niej modyfikowana tak, że efektem jest tablica posortowana rosnąco.	
7	void selectionSort (int *arr, int n)	selectionSort	void	int *arr: wskaźnik na tablicę liczb całkowitych int n: liczba elementów tablicy	Jest to implementacja algorytmu sortowania przez wstawianie. Tablica, której wskaźnik jest argumentem funkcji, jest w niej modyfikowana tak, że efektem jest tablica posortowana rosnąco.	Funkcja wywołuje inną funkcję w swoim ciele – swap.
8	void quickSort(int *arr, int left, int right)	quickSort	void	int *arr: wskaźnik na tablicę liczb całkowitych int left: indeks pierwszego elementu tablicy int right: indeks ostatniego elementu tablicy	Jest to rekurencyjna funkcja będąca implementacją algorytmu sortowania szybkiego. Tablica, której wskaźnik jest argumentem funkcji, jest w niej modyfikowana tak, że efektem jest tablica posortowana rosnąco od left do right.	Funkcja wywołuje inną funkcję w swoim ciele – partition.
9	int partition(int *arr, int left, int right)	partition	int j: funkcja zwraca indeks j zmodyfikowanej tablicy arr; indeks ten jest ostatnim, dla którego wartość arr[j] jest nie większa od elementu arr[left]	int *arr: wskaźnik na tablicę liczb całkowitych int left: indeks pierwszego elementu tablicy int right: indeks ostatniego elementu tablicy	Tablica arr jest modyfikowana wewnątrz funkcji w taki sposób, że elementy między indeksami left i j są nie większe od arr[left]. Funkcja następnie zwraca indeks j.	Funkcja wywołuje inną funkcję w swoim ciele – swap. Jest wywoływana wyłącznie przez funkcję quickSort.
10	void heapSort(int *arr, int n)	heapSort	void	int *arr: wskaźnik na tablicę liczb całkowitych int n: liczba elementów tablicy	Jest główna funkcja implementacji sortowania przez kopcowanie. Tablica, której wskaźnik jest argumentem funkcji, jest w niej modyfikowana tak, że efektem jest tablica posortowana rosnąco.	Funkcja wywołuje w swoim ciele 2 inne funkcje – heapify i buildHeap.
11	void buildHeap(int *arr, int n)	buildHeap	void	int *arr: wskaźnik na tablicę liczb całkowitych int n: liczba elementów tablicy	Funkcja odpowiada za budowanie struktury kopca.	Funkcja wywołuje w swoim ciele inną funkcję – heapify.
12	void heapify(int *arr, int heap_size, int i)	heapify	void	int *arr: wskaźnik na tablicę liczb całkowitych int heap_size: rozmiar kopca	Funkcja odpowiada za zachowanie prawidłowej struktury kopca binarnego typu max. Ma ona	Funkcja wywołuje w swoim ciele inną

				int j: indeks elementu kopca, którego dzieci będą sprawdzane pod względem poprawności	charakter rekurencyjny. Modyfikuje w swoim wnętrzu tablicę arr.	funkcją – swap.
13	void countingSort(int *arr, int n)	countingSort	void	int *arr: wskaźnik na tablicę liczb całkowitych nieujemnych int n: liczba elementów tablicy	Jest to implementacja sortowania przez zliczanie. Tablica, której wskaźnik jest argumentem funkcji, jest w niej modyfikowana tak, że efektem jest tablica posortowana rosnąco.	Funkcja wywołuje w swoim ciele inną funkcję - free2DArray oraz maxIndex.
14	int maxIndex(int *arr, int n)	maxIndex	int max_index: indeks największego elementu tablicy arr	int *arr: wskaźnik na tablicę liczb całkowitych int n: liczba elementów tablicy	Tablica jest skanowana liniowo i zwracany jest indeks największego jej elementu.	-
15	void free2DArray (int **arr, int rows)	free2DArray	void	int **arr: wskaźnik na macierz int rows: liczba wierszów macierzy	Funkcja odpowiada za zwalnianie sterty (heap) po macierzy.	-
16	void print2DArray(int **arr, int n, int m)	print2DArray	void	int **arr: wskaźnik na macierz int n: liczba wierszów macierzy int m: liczba kolumn macierzy	Funkcja drukuje na standardowe wyjście macierz.	-
17	int *generateRandomNumbers(int maxNum, int elements)	generateRandomNumbers	int*arr: wskaźnik na tablicę liczb całkowitych	int maxNum: maksymalna możliwa wartość w tablicy int elements: liczba elementów tablicy	Funkcja alokuje miejsca w stercie (heap) na tablicę, wypełnia ją liczbami elements pseudolosowymi nie większymi niż maxNum i zwraca wskaźnik na tę tablicę.	Po jej wywołaniu, należy zwolnić pamięć w stercie.
18	void saveToFile(FILE *stream, int *arr, int elements)	saveToFile	void	FILE *stream: wskaźnik na plik int *arr: wskaźnik na tablicę int elements: liczba elementów tablicy	Funkcja zapisuje tablicę arr w pliku, wskazywanym przez stream.	Ta funkcja modyfikuje plik zewnętrzny.
19	int *copyArray(int *orgarr, int n)	copyArray	int *copied: wskaźnik na tablicę liczb całkowitych	int *orgarr: tablica wejściowa liczb całkowitych int n: liczba elementów tablicy	Funkcja alokuje miejsce w stercie, kopiuje tablicę orgarr i zwraca wskaźnik na skopiowaną tablicę.	Po jej wywołaniu, należy zwolnić pamięć w stercie.
20	int *reverseSort(int *arr, int n)	reverseSort	int *rev: wskaźnik na tablicę liczb całkowitych	int *arr: wskaźnik na tablicę liczb całkowitych int n: liczba elementów tablicy	Funkcja alokuje miejsce w stercie i zwraca wskaźnik na tablicę będącą przeciwnie posortowaną tablicą arr.	Funkcja wywołuje w swoim ciele inną funkcję - copyArray oraz mergeSort. Po jej wywołaniu, należy zwolnić pamięć w stercie.



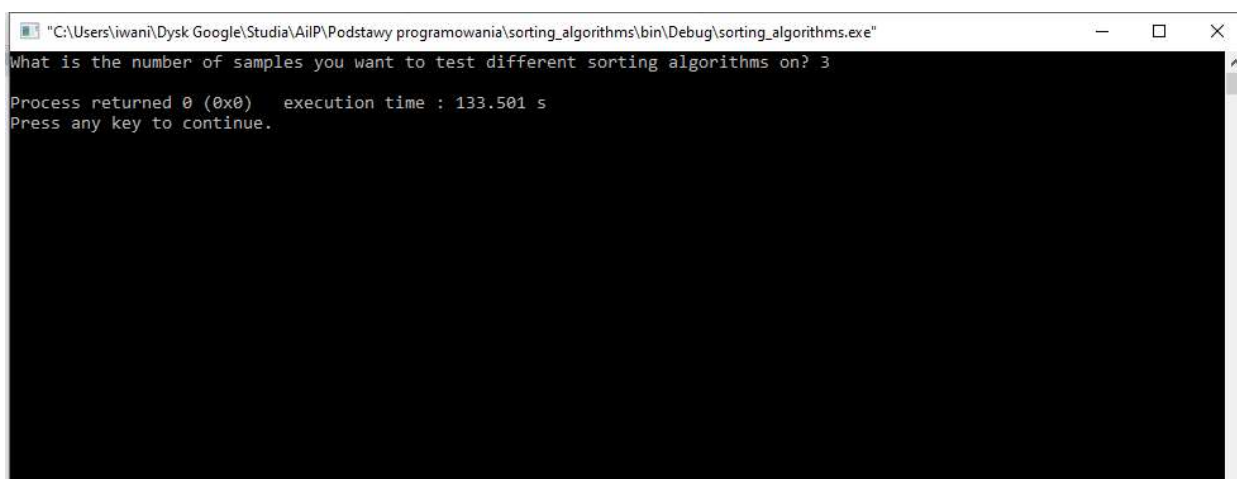
## 6. Dokumentacja działania programu

Istnieją 3 miejsca, w których użytkownik może spersonalizować działanie programu:

- 1) Ustawienie wartości dyrektywy MAX\_NUM. Im większa, tym większe mogą być wartości generowane w tablicy. Należy pamiętać o tym, że nie powinna być większa niż maksymalna możliwa wartość nadawana 4-bajtowej signed int (czyli 2 147 483 647).
- 2) Ustawienie wartości dyrektywy MAX\_SAMPLE\_SIZE. Mówi ona o maksymalnym możliwym rozmiarze tablicy liczb pseudolosowych.
- 3) Po uruchomieniu pliku wykonywalnego, użytkownik może podać liczbę tablic próbnych, które będą następnie poddawane sortowaniu.

Pierwsze dwa ograniczenia zostały dobrane tak, by umożliwić zakończenie obliczeń w skali minut. Trzeci parametr może być zmieniany dowolnie, zależnie od ilości danych, które chcemy zebrać.

Przykładowy zrzut ekranu:



```
"C:\Users\iwani\Disk Google\Studia\AIIP\Podstawy programowania\sorting_algorithms\bin\Debug\sorting_algorithms.exe"
What is the number of samples you want to test different sorting algorithms on? 3
Process returned 0 (0x0)   execution time : 133.501 s
Press any key to continue.
```

Efekt działania programu możemy zbadać w pliku tekstowym „data.txt”, utworzonym w katalogu z plikiem .exe. Jego zawartość jest przy uruchomieniu programu nadpisywana. Plik po uruchomieniu programu zawiera następujące informacje:

- a) wskazaną przez użytkownika liczbę tablic próbnych
- b) numer iteracji
- c) wygenerowaną przez program maksymalną dopuszczalną w danej iteracji wartość tablicy (która oczywiście podlega jeszcze ograniczeniu dyrektywy MAX\_NUM) i wielkość tablicy
- d) nazwę algorytmu sortującego wraz z względnym czasem sortowania

```

1 arrays_number = 3    iteration = 1
2 max_number = 1205167    array_size = 62409
3
4 Random array
5 bubbleSort    17.118000000000
6 insertionSort    2.353000000000
7 selectionSort    4.512000000000
8 heapSort    0.015000000000
9 countingSort    0.000000000000
10 mergeSort    0.016000000000
11 quickSort    0.007000000000
12
13 Sorted array
14 bubbleSort    0.000000000000
15 insertionSort    0.000000000000
16 selectionSort    4.712000000000
17 heapSort    0.016000000000
18 countingSort    0.000000000000
19 mergeSort    0.015000000000
20 quickSort    2.074000000000
21
22 Reversed array
23 bubbleSort    16.359000000000
24 insertionSort    4.744000000000
25 selectionSort    7.918000000000
26 heapSort    0.008000000000
27 countingSort    0.012000000000
28 mergeSort    0.010000000000
29 quickSort    4.338000000000
30
31 -----
32 arrays_number = 3    iteration = 2
33 max_number = 1835056    array_size = 25685
34
35 Random array
36 bubbleSort    2.726000000000
37 insertionSort    0.394000000000
38 selectionSort    0.752000000000
39 heapSort    0.016000000000
40 countingSort    0.000000000000
41 mergeSort    0.000000000000
42 quickSort    0.015000000000

```

## 7. Kod źródłowy

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// MAX_NUM is the maximum possible value of a number generated in a random array
#define MAX_NUM 2000000
// MAX_SAMPLE_SIZE is the maximum possible length of a random array
#define MAX_SAMPLE_SIZE 100000
// MAX_CHAR is the longest possible function name (as stored in "functions" struct)
#define MAX_CHAR 30
// SORT_ALG_N is the number of sorting algorithms which take as an argument a
// pointer to an array and the number of elements
#define SORT_ALG_N 5
// SORT_ALG_ST is the number of sorting algorithms which take as an argument a
// pointer to an array as well as the first and the last index of an array
#define SORT_ALG_ST 2

void mergeSort(int *arr, int start, int finish);
void merge(int *arr, int start, int middle, int finish);
void printArray(int *arr, int n);
void bubbleSort(int *arr, int n);
void swap (int *a, int *b);
void insertionSort(int *arr, int n);
void selectionSort (int *arr, int n);
void quickSort(int *arr, int left, int right);
int partition(int *arr, int left, int right);
void heapSort(int *arr, int n);
void buildHeap(int *arr, int n);
void heapify(int *arr, int heap_size, int i);
int maxIndex(int *arr, int n);
void countingSort(int *arr, int n);
void free2DArray (int **arr, int rows);
void print2DArray(int **arr, int n, int m);
int *generateRandomNumbers(int maxNum, int elements);
void saveToFile(FILE *stream, int *arr, int elements);
int *copyArray(int *orgarr, int n);
int *reverseSort(int * arr, int n);

// type definition: a pointer to a function which takes 2 arguments: an array of
// integers and an array size (an integer)
typedef void (*ARR_N)(int*,int);
// type definition: a pointer to a function which takes 3 arguments: an array of
// integers, first index, last index
typedef void (*ARR_ST)(int*,int,int);

// "functions" struct contains arrays of pointers to sorting functions and arrays
// of their names
struct functions
{
    ARR_N fun_arrN[SORT_ALG_N];
    ARR_ST fun_arrST[SORT_ALG_ST];
    char names_fun_arrN[SORT_ALG_N][MAX_CHAR];
    char names_fun_arrST[SORT_ALG_ST][MAX_CHAR];
};

int main()
{
    // a pointer to a FILE variable
    FILE *stream;

    // initialization of "functions" struct
    // two arrays contain elements that point to sorting functions
```

```

    struct functions funstruct = { {bubbleSort, insertionSort, selectionSort,
heapSort, countingSort}, {mergeSort, quickSort}},
{"bubbleSort", "insertionSort", "selectionSort", "heapSort", "countingSort"},
{"mergeSort", "quickSort"} };

    // i, j - counter variables
    // arrays_number - the number of arrays containing pseudo-random numbers
    // seed - variable used to create seed for generation of pseudo-random numbers
    // arr - an array of pseudo-random numbers
    // rev is an array containing the same values as arr, in decreasing order
    // copied contains a copy of an array that sorting functions work on
    // arrsor is an array containing the same values as arr, in increasing order
    // max_number variable stores largest possible number that can occur in "arr",
as generated by rand (compare: MAX_NUM is an upper possible limit)
    // array_size stores an array size, as generated by rand
    int i, j, arrays_number, seed, *arr, *rev, *copied, *arrsor, max_number,
array_size;

    // auxiliary variable to generate PRN
    time_t t;

    // for counting execution time
    clock_t start, end;

    // time of execution in seconds
    double seconds;

    // initialization of pseudo-random number generator
    seed = time(&t);
    srand(seed);

    // opening of a file
    stream = fopen("data.txt", "w+");

    // error checking
    if (stream == NULL) {
        printf("File error.");
        return -1;
    }

    printf("What is the number of samples you want to test different sorting
algorithms on?\t");
    scanf("%d", &arrays_number);

    for (i=0; i<arrays_number; i++) {
        // in each iteration, max_number and array_size will be (pseudo)-randomly
generated
        max_number = ((rand() * rand()) % MAX_NUM) + 1;
        array_size = ((rand() * rand()) % MAX_SAMPLE_SIZE) + 1;

        fprintf(stream, "arrays_number = %d\titeration = %d\nmax_number = %d\
tarray_size = %d\n", arrays_number, i+1, max_number, array_size);

        // 3 types of arrays are sorted
        // 1) (pseudo)-random array
        // 2) sorted array (here it is sorted by bubbleSort, but it does not
matter)
        // 3) "reversed" array (decreasing order)
        arr = generateRandomNumbers(max_number, array_size);
        arrsor = copyArray(arr, array_size);
        bubbleSort(arrsor, array_size);
        rev = reverseSort(arr, array_size);

        // in each case (random, sorted, reversed), the timer starts at the
function call and ends as it falls off the stack
        // the time is stored in variable seconds
        // the result is printed, along with algorithm's name, to a file pointed to
by stream

```

```

fprintf(stream, "\nRandom array\n");
for (j=0; j<SORT_ALG_N; j++) {
    copied = copyArray(arr,array_size);
    start = clock();
    funstruct.fun_arrN[j](copied,array_size);
    end = clock();
    seconds = (double)(end - start) / CLOCKS_PER_SEC;
    fprintf(stream, "%s\t\t%.12f\n", funstruct.names_fun_arrN[j], seconds);
    free(copied);
}

for (j=0; j<SORT_ALG_ST; j++) {
    copied = copyArray(arr,array_size);
    start = clock();
    funstruct.fun_arrST[j](copied,0,array_size-1);
    end = clock();
    seconds = (double)(end - start) / CLOCKS_PER_SEC;
    fprintf(stream, "%s\t\t%.12f\n", funstruct.names_fun_arrST[j], seconds);
    free(copied);
}

fprintf(stream, "\nSorted array\n");
for (j=0; j<SORT_ALG_N; j++) {
    copied = copyArray(arrsor,array_size);
    start = clock();
    funstruct.fun_arrN[j](copied,array_size);
    end = clock();
    seconds = (double)(end - start) / CLOCKS_PER_SEC;
    fprintf(stream, "%s\t\t%.12f\n", funstruct.names_fun_arrN[j], seconds);
    free(copied);
}

for (j=0; j<SORT_ALG_ST; j++) {
    copied = copyArray(arrsor,array_size);
    start = clock();
    funstruct.fun_arrST[j](copied,0,array_size-1);
    end = clock();
    seconds = (double)(end - start) / CLOCKS_PER_SEC;
    fprintf(stream, "%s\t\t%.12f\n", funstruct.names_fun_arrST[j], seconds);
    free(copied);
}

fprintf(stream, "\nReversed array\n");
for (j=0; j<SORT_ALG_N; j++) {
    copied = copyArray(rev,array_size);
    start = clock();
    funstruct.fun_arrN[j](copied,array_size);
    end = clock();
    seconds = (double)(end - start) / CLOCKS_PER_SEC;
    fprintf(stream, "%s\t\t%.12f\n", funstruct.names_fun_arrN[j], seconds);
    free(copied);
}

for (j=0; j<SORT_ALG_ST; j++) {
    copied = copyArray(rev,array_size);
    start = clock();
    funstruct.fun_arrST[j](copied,0,array_size-1);
    end = clock();
    seconds = (double)(end - start) / CLOCKS_PER_SEC;
    fprintf(stream, "%s\t\t%.12f\n", funstruct.names_fun_arrST[j], seconds);
    free(copied);
}

fprintf(stream, "\n-----\n");

free(arrsor);
free(rev);
free(arr);

```

```

    }

    fclose(stream);
    return 0;
}

// the function takes 4 arguments: an array; start, middle and finish indeces
// arr has to sorted in the following way: increasingly from start to middle,
// increasingly from middle+1 to finish
// it modifies array arr
// it modifies an array so that it is sorted increasingly from start to finish
void merge(int *arr, int start, int middle, int finish)
{
    int *p, i1, i2, k;
    p = (int*)malloc(sizeof(int)*(finish-start+1));

    i1 = start;
    i2 = middle + 1;
    k = 0;

    while (i1<=middle && i2<=finish) {
        if (arr[i1] <= arr[i2])
            p[k] = arr[i1++];
        else
            p[k] = arr[i2++];
        k++;
    }

    while (i1<=middle) {
        p[k] = arr[i1++];
        k++;
    }

    while (i2<=finish) {
        p[k] = arr[i2++];
        k++;
    }

    for (k=0; k<=finish-start; k++)
        arr[start+k] = p[k];

    free(p);
}

// this function is a main divide-and-conquer recursive function of a merge sort
// algorithm
// it takes 3 arguments: an array, first and last indeces of an array
// it return nothing
// it modifies array arr
void mergeSort(int *arr, int start, int finish)
{
    int middle;
    if (start!=finish) {
        middle = (int)((start+finish)/2);
        mergeSort(arr, start, middle);
        mergeSort(arr, middle+1, finish);
        merge(arr, start, middle, finish);
    }
}

// the function prints array arr
// it takes 2 arguments: an array and its length
// it return nothing
void printArray(int *arr, int n)
{
    for (int i =0; i<n; i++) {

```

```

        printf("%d ", arr[i]);
    }
}

// this function sorts an array using bubble sort algorithm
// it modifies array arr
// it return nothing
// it takes 2 arguments: an array and its length
void bubbleSort(int *arr, int n)
{
    int i,k=1;

    while (k) {
        k = 0;
        for (i=n-2; i>=0; i--) {
            if (arr[i+1] < arr[i]) {
                swap(&arr[i+1], &arr[i]);
                k++;
                //printf("k = %d\n", k);
            }
        }
    }
}

// the function swap values of two variables
// it takes 2 arguments: a pointer to variable a, a pointer to variable b
// it return nothing
// it modifies original variables, as pointers to them are passed to the function
void swap (int *a, int *b)
{
    int c;
    c = *a;
    *a = *b;
    *b = c;
}

// the function sorts an array using insertion sort algorithm
// it takes 2 arguments: an array and its length
// it return nothing
// it modifies array arr
void insertionSort(int *arr, int n)
{
    int i, hole, value;
    for (i=1; i<n; i++) {
        value = arr[i];
        hole = i;
        while (hole > 0 && (arr[hole-1] > value)) {
            arr[hole] = arr[hole-1];
            hole --;
        }
        arr[hole] = value;
    }
}

// the function sorts an array using selection sort algorithm
// it takes 2 arguments: an array and its length
// it return nothing
// it modifies array arr
void selectionSort (int *arr, int n)
{
    int i, j, min_index;

    for (i=0; i<n-1; i++) {
        min_index = i;
        for (j=i+1; j<n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
    }
}

```

```

    }
    swap(&arr[min_index], &arr[i]);
}

// this function is a main divide-and-conquer recursive function of a quick sort
algorithm
// it takes 3 arguments: an array, first and last indeces of an array
// it returns nothing
// it modifies array arr
void quickSort(int *arr, int left, int right)
{
    int m;
    if (left >=right)
        return;

    m = partition(arr, left, right);
    quickSort(arr, left, m-1);
    quickSort(arr, m+1, right);
}

// this function is used by quickSort function
// it takes 3 arguments: an array, first and last indeces of an array
// it returns an index of the last element not greater than arr[left]
// it also modifies array arr
int partition(int *arr, int left, int right)
{
    int x, j, i;
    x = arr[left];
    j = left;

    for (i=left+1; i<=right; i++) {
        if (arr[i] <= x) {
            j++;
            swap(&arr[j], &arr[i]);
        }
    }

    swap(&arr[left], &arr[j]);

    return j;
}

// this function is a main function of a heap sort algorithm
// it takes 2 arguments: an array and its length
// it return nothing
// it modifies array arr; it calls 2 other functions which in turn also modify
array arr
void heapSort(int *arr, int n)
{
    int i;
    buildHeap(arr, n);
    for (i=n-1; i>0; i--) {
        swap(&arr[0], &arr[i]);
        n--;
        heapify(arr, n, 0);
    }
}

// the function builds heap, a structure using in heap sort algorithm
// it takes 2 arguments: an array and its length
// it return nothing
// it calls 1 other function which in turn modifies array arr
void buildHeap(int *arr, int n)
{
    int i;
    for (i=n/2; i>=0; i--) {
        heapify(arr, n, i);
    }
}

```



```

    }
}

// the function makes sure that the heap has a proper property
// it takes 3 arguments: an array, heap size and index i
// it return nothing
// it modifies array arr
void heapify(int *arr, int heap_size, int i)
{
    int largest, right, left;
    left = 2*i + 1;
    right = 2*i + 2;
    largest = i;

    if (left < heap_size && arr[left]>arr[largest])
        largest = left;
    if (right < heap_size && arr[right] > arr[largest])
        largest = right;
    if (largest != i) {
        swap(&arr[largest], &arr[i]);
        heapify(arr, heap_size, largest);
    }
}

// the function sorts array arr using counting sort algorithm
// it can only sort non-negative integers
// it takes 2 arguments: array and its length
// it return nothing
// it modifies array arr
void countingSort(int *arr, int n)
{
    int **p, i, p_size, k, p_ind;
    p_size = arr[maxIndex(arr,n)] + 1;
    p = (int**)malloc(sizeof(int) * p_size);

    for (i=0; i<p_size; i++)
        p[i] = (int*)malloc(sizeof(int) * 2);

    for (i=0; i<p_size; i++) {
        p[i][0] = i;
        p[i][1] = 0;
    }

    for (i=0; i<n; i++) {
        p_ind = arr[i];
        p[p_ind][1] += 1;
    }

    i = 0;
    k = 0;
    while (i<p_size) {
        while (p[i][1]!=0) {
            arr[k] = p[i][0];
            k++;
            p[i][1] -= 1;
        }
        i++;
    }

    free2DArray(p, p_size);
}

// the function find an index of largest element in an array
// it takes 2 arguments: array and its length
// it return an index of the largest element
int maxIndex(int *arr, int n)

```

```

{
    int i, max_index=0;
    for (i=1; i<n; i++) {
        if (arr[max_index] < arr[i])
            max_index = i;
    }
    return max_index;
}

// the functions frees RAM of a 2D array
// it takes 2 arguments - an array and the number of its columns
// it returns nothing
void free2DArray (int **arr, int rows)
{
    int i;
    for (i=0; i<rows; i++)
        free(arr[i]);
    free(arr);
}

// the functions prints 2D array
// it takes 3 arguments: an array, the number of rows, the number of columns
// it returns nothing
void print2DArray(int **arr, int n, int m)
{
    int i, j;
    for (i=0; i<n; i++) {
        for (j=0; j<m; j++) {
            printf("%d\t", arr[i][j]);
        }
        printf("\n");
    }
}

// the function generates "elements" number of random integers from (0...maxNum)
// this function takes
// maxN - maximum number possible (we will generate numbers from 0 to maxNum)
// element - number of random number in a file
// it return a pointer to an array
int *generateRandomNumbers(int maxNum, int elements)
{
    int i, *arr;

    arr = (int*)malloc(sizeof(int) * elements);

    for (i=0; i<elements; i++) {
        arr[i] = rand() % (maxNum+1);
    }

    return arr;
}

// the function saves an array to a file
// it takes 3 arguments: pointer to a file, an array, its length
// it returns nothing
// it modifies a file pointed to by a *stream variable
void saveToFile(FILE *stream, int *arr, int elements)
{
    int i;
    for (i=0; i<elements-1; i++) {
        fprintf(stream, "%d, ", arr[i]);
    }
    fprintf(stream, "%d", arr[i]);
}

// the function copies an array to another array
// it takes 2 arguments: an array to be copied and its length
// it return a pointer to a copied array

```

```

int *copyArray(int *orgarr, int n)
{
    int *copied, i;

    copied = (int*)malloc(sizeof(int)*n);

    for (i=0; i<n; i++) {
        copied[i] = orgarr[i];
    }

    return copied;
}

// the function returns reverse sorted arr
// it takes 2 arguments: an array and its length
// it return a pointer to an array (reverse-sorted)
int *reverseSort(int * arr, int n)
{
    // rev is reversed array
    // sor is sorted array/
    // n - length of array
    // i, k iterator
    int *rev, *sor, i, k;

    rev = (int*)malloc(sizeof(int)*n);
    sor = (int*)malloc(sizeof(int)*n);

    sor = copyArray(arr, n);
    mergeSort(sor, 0, n-1);

    i = 0;
    k = n-1;
    while (i<n) {
        rev[i] = sor[k];
        i++;
        k--;
    }

    free(sor);

    return rev;
}

```