Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КОМП'ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Катедра «Комп'ютерна інженерія та програмування»

ЗВІТ

про виконання лабораторної роботи №12

з навчальної дисципліни «Алгоритми та структури даних»

Варіант 9

Виконав студент:

Ульянов Кирило Юрійович
Група: КН-1023b

Перевірив:

старший викладач

Бульба С.С.

Харків-2024

1 Мета роботи

Закріпити теоретичні знання та набути практичного досвіду впорядкування набору статичних та динамічних структур даних.

2 Хід роботи

Написати програму, що реалізує три алгоритми сортування набору даних згідно з табл. 12.1.

Визначити кількість порівнянь та обмінів для початкових наборів даних, що містять різну кількість елементів (50, 1000, 5000, 10000, 50000).

Оцінити час сортування. Дослідити вплив початкової впорядкованості набору даних (відсортований, відсортований у зворотному порядку, випадковий).

Всі отримані дані записати в табл. 12.2. Зробити висновки.

Моє завдання:
Алгоритми сортування:

- сортування частково впорядкованим деревом
- порозрядне цифрове сортування
- сортування прямим злиттям

Массив: своя структура динамічного массиву

2.1 Реалізація динамічного масиву

У структурі було створено методи: *ініціалізації, знищення, додавання елементів, отримання елементів*.

Файл **заголовків**

```h
#ifndef DYN_ARR_H
#define DYN_ARR_H

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

typedef struct DynArr
{
  void *data;
  size_t elem_size;
  size_t size;
  size_t capacity;
} DynArr;

DynArr *init_dyn_arr(size_t element_size, size_t initial_capacity);
void push_dyn_arr(DynArr *array, void *element);
void *get_elem_dyn_arr(DynArr *array, size_t index);
void destroy_dyn_arr(DynArr *array);

#endif
```

Файл **реалізації**

```c
#include "dyn_arr.h"

// init arr
DynArr *init_dyn_arr(size_t element_size, size_t initial_capacity)
{
  DynArr *arr = malloc(sizeof(DynArr));
  if (!arr)
  {
    perror("Failed to allocate memory for array");
    exit(EXIT_FAILURE);
  }

  arr->data = calloc(element_size, initial_capacity);
  if (!arr->data)
  {
    perror("Failed to allocate memory for array data");
    free(arr);
    exit(EXIT_FAILURE);
  }

  arr->elem_size = element_size;
  arr->size = 0;
  arr->capacity = initial_capacity;

  return arr;
}

// resize arr
```

```c
void _resize_arr(DynArr *arr, size_t new_capacity)
{
  arr->data = realloc(arr->data, arr->elem_size * new_capacity);
  if (!arr->data)
  {
    perror("Failed to reallocate memory");
    exit(EXIT_FAILURE);
  }
  arr->capacity = new_capacity;
}

// push elem to arr
void push_dyn_arr(DynArr *arr, void *element)
{
  if (arr->size == arr->capacity)
  {
    _resize_arr(arr, arr->capacity * 2);
  }

  memcpy((char *)arr->data + arr->size * arr->elem_size, element, arr->
      elem_size);
  arr->size++;
}

// get elem by index from arr
void *get_elem_dyn_arr(DynArr *arr, size_t index)
{
  if (index >= arr->size)
  {
    fprintf(stderr, "Index out of bounds\n");
    return NULL;
  }
  return (char *)arr->data + index * arr->elem_size;
}

// destroy arr
void destroy_dyn_arr(DynArr *arr)
{
  free(arr->data);
  free(arr);
}
```

2.2 Реалізація сортувань

Всі сортування було створено та об‘єднано спеціальною структурою для того щоб виводити однакові метрики для всі сортувань. Кожна функція приймає структуру динамічного масиву та порядок сортування та виведе на екран кількість перестановок та порівнянь. Окрім сортування частоково впорядкованими деревами, яке ще приймає функцію порівняння.

2.2.1 Реалізація алгоритму сортування злиттям

```c
#include "merge_sort.h"

int compare_elements(void *a, void *b, size_t elem_size, SortOrder order)
{
  if (order == ASCENDING)
  {
    return memcmp(a, b, elem_size);
  }
  else
  {
    return memcmp(b, a, elem_size);
  }
}

void merge(DynArr *arr, size_t left, size_t mid, size_t right, SortOrder order, SortStats *stats)
{
  size_t left_size = mid - left + 1;
  size_t right_size = right - mid;

  void *left_arr = malloc(left_size * arr->elem_size);
  void *right_arr = malloc(right_size * arr->elem_size);

  if (!left_arr || !right_arr)
  {
    perror("Failed to allocate memory for merge");
    exit(EXIT_FAILURE);
  }

  memcpy(left_arr, (char *)arr->data + left * arr->elem_size, left_size * arr->elem_size);
  memcpy(right_arr, (char *)arr->data + (mid + 1) * arr->elem_size, right_size * arr->elem_size);

  size_t i = 0, j = 0, k = left;

  while (i < left_size && j < right_size)
  {
    stats->comparisons++;
    if (compare_elements((char *)left_arr + i * arr->elem_size, (char *)right_arr + j * arr->elem_size, arr->elem_size, order) <= 0)
    {
      memcpy((char *)arr->data + k * arr->elem_size, (char *)left_arr + i * arr->elem_size, arr->elem_size);
      i++;
```

```
41        }
42        else
43        {
44          memcpy((char *)arr->data + k * arr->elem_size, (char *)right_arr +
                j * arr->elem_size, arr->elem_size);
45          j++;
46        }
47        k++;
48        stats->swaps++;
49      }
50
51      while (i < left_size)
52      {
53        memcpy((char *)arr->data + k * arr->elem_size, (char *)left_arr + i
              * arr->elem_size, arr->elem_size);
54        i++;
55        k++;
56        stats->swaps++;
57      }
58
59      while (j < right_size)
60      {
61        memcpy((char *)arr->data + k * arr->elem_size, (char *)right_arr + j
              * arr->elem_size, arr->elem_size);
62        j++;
63        k++;
64        stats->swaps++;
65      }
66
67      free(left_arr);
68      free(right_arr);
69    }
70
71    void merge_sort(DynArr *arr, size_t left, size_t right, SortOrder order,
          SortStats *stats)
72    {
73      if (left < right)
74      {
75        size_t mid = left + (right - left) / 2;
76
77        merge_sort(arr, left, mid, order, stats);
78        merge_sort(arr, mid + 1, right, order, stats);
79        merge(arr, left, mid, right, order, stats);
80      }
81    }
82
83    void merge_sort_dyn_arr(DynArr *arr, SortOrder order)
84    {
85      SortStats stats = {0, 0};
86      merge_sort(arr, 0, arr->size - 1, order, &stats);
87
88      if (order == ASCENDING)
89      {
90        printf("\n\033[32 m\033[0m Sorting order: \033[32masc\033[0m\n");
91      }
92      else
93      {
94        printf("\n\033[32 m\033[0m Sorting order: \033[32mdesc\033[0m\n");
95      }
96
```

```
97      printf("\033[32 m\033[0m Number of swaps: \033[32m%d\033[0m\n", stats.
            swaps);
98      printf("\033[32 m\033[0m Number of comparisons: \033[32m%d\033[0m\n",
            stats.comparisons);
99    }
```

2.2.2 Реалізація алгоритму порозрядного цифрового сортування

```
1   #include "radix_sort.h"
2
3   int get_digit(int number, int place)
4   {
5     return (number / place) % 10;
6   }
7
8   int get_max(DynArr *arr)
9   {
10    int max = *(int *)get_elem_dyn_arr(arr, 0);
11    for (size_t i = 1; i < arr->size; i++)
12    {
13      int value = *(int *)get_elem_dyn_arr(arr, i);
14      if (value > max)
15      {
16        max = value;
17      }
18    }
19    return max;
20  }
21
22  void counting_sort(DynArr *arr, int place, SortOrder order, SortStats *
        stats)
23  {
24    size_t count[10] = {0};
25    int *output = malloc(arr->size * sizeof(int));
26
27    if (!output)
28    {
29      perror("Failed to allocate memory for output array");
30      exit(EXIT_FAILURE);
31    }
32
33    for (size_t i = 0; i < arr->size; i++)
34    {
35      int digit = get_digit(*(int *)get_elem_dyn_arr(arr, i), place);
36      count[digit]++;
37    }
38
39    if (order == ASCENDING)
40    {
41      for (int i = 1; i < 10; i++)
42      {
43        count[i] += count[i - 1];
44      }
45    }
46    else
47    {
48      for (int i = 8; i >= 0; i--)
```

```
49      {
50        count[i] += count[i + 1];
51      }
52    }
53
54    for (int i = arr->size - 1; i >= 0; i--)
55    {
56      int digit = get_digit(*(int *)get_elem_dyn_arr(arr, i), place);
57      output[--count[digit]] = *(int *)get_elem_dyn_arr(arr, i);
58      stats->swaps++;
59    }
60
61    for (size_t i = 0; i < arr->size; i++)
62    {
63      memcpy((char *)arr->data + i * arr->elem_size, &output[i], sizeof(int)
          );
64    }
65
66    free(output);
67  }
68
69  void radix_sort(DynArr *arr, SortOrder order, SortStats *stats)
70  {
71    int max = get_max(arr);
72
73    for (int place = 1; max / place > 0; place *= 10)
74    {
75      counting_sort(arr, place, order, stats);
76      stats->comparisons += arr->size;
77    }
78  }
79
80  void radix_sort_dyn_arr(DynArr *arr, SortOrder order)
81  {
82    SortStats stats = {0, 0};
83    radix_sort(arr, order, &stats);
84
85    if (order == ASCENDING)
86    {
87      printf("\n\033[32 m\033[0m Sorting order: \033[32masc\033[0m\n");
88    }
89    else
90    {
91      printf("\n\033[32 m\033[0m Sorting order: \033[32mdesc\033[0m\n");
92    }
93
94    printf("\033[32 m\033[0m Number of swaps: \033[32m%d\033[0m\n", stats.
        swaps);
95    printf("\033[32 m\033[0m Number of comparisons: \033[32m%d\033[0m\n",
        stats.comparisons);
96  }
```

2.2.3 Реалізація алгоритму сортування частково впорядкованим деревом

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "heap_sort.h"

//
static void swap_elements(DynArr *arr, size_t i, size_t j, SortStats *stats)
{
  if (i == j)
    return;
  //
  void *temp = malloc(arr->elem_size);
  if (!temp)
  {
    perror("Failed to allocate memory for swap");
    exit(EXIT_FAILURE);
  }

  void *elem_i = (char *)arr->data + i * arr->elem_size;
  void *elem_j = (char *)arr->data + j * arr->elem_size;

  memcpy(temp, elem_i, arr->elem_size);
  memcpy(elem_i, elem_j, arr->elem_size);
  memcpy(elem_j, temp, arr->elem_size);

  free(temp);

  stats->swaps++;
}

//
static int compare_elements(const void *a, const void *b, int (*compare)(const void *, const void *), SortStats *stats)
{
  stats->comparisons++;
  return compare(a, b);
}

//                (heapify)
//                 -     max-heap    (           ).
//                 -     min-heap    (           ).
static void heapify(DynArr *arr, size_t n, size_t i, SortOrder order, int (*compare)(const void *, const void *), SortStats *stats)
{
  size_t largest_or_smallest = i;
  size_t left = 2 * i + 1;
  size_t right = 2 * i + 2;

  //           ,
  //    ASCENDING:       max-heap,                    .
  // largest_or_smallest
  //    DESCENDING:       min-heap,                    .
  // largest_or_smallest        .

  //
  if (left < n)
```

```
55    {
56      void *current = (char *)arr->data + largest_or_smallest * arr->
            elem_size;
57      void *child = (char *)arr->data + left * arr->elem_size;
58      int cmp = compare_elements(child, current, compare, stats);
59
60      if ((order == ASCENDING && cmp > 0) || (order == DESCENDING && cmp <
            0))
61      {
62        largest_or_smallest = left;
63      }
64    }
65
66    //
67    if (right < n)
68    {
69      void *current = (char *)arr->data + largest_or_smallest * arr->
            elem_size;
70      void *child = (char *)arr->data + right * arr->elem_size;
71      int cmp = compare_elements(child, current, compare, stats);
72
73      if ((order == ASCENDING && cmp > 0) || (order == DESCENDING && cmp <
            0))
74      {
75        largest_or_smallest = right;
76      }
77    }
78
79    //             /      ,
80    if (largest_or_smallest != i)
81    {
82      swap_elements(arr, i, largest_or_smallest, stats);
83      heapify(arr, n, largest_or_smallest, order, compare, stats);
84    }
85  }
86
87  void heap_sort_dyn_arr(DynArr *arr, SortOrder order, int (*compare)(const
        void *, const void *))
88  {
89    SortStats stats = {0, 0};
90
91    size_t n = arr->size;
92    if (n < 2)
93    {
94      //
95      if (order == ASCENDING)
96      {
97        printf("\n\033[32 m\033[0m Sorting order: \033[32masc\033[0m\n");
98      }
99      else
100     {
101       printf("\n\033[32 m\033[0m Sorting order: \033[32mdesc\033[0m\n");
102     }
103
104     printf("\033[32 m\033[0m Number of swaps: \033[32m%zu\033[0m\n", stats.
            swaps);
105     printf("\033[32 m\033[0m Number of comparisons: \033[32m%zu\033[0m\n",
            stats.comparisons);
106     return;
107   }
```

```
108
109     //
110     for (int i = (int)(n / 2) - 1; i >= 0; i--)
111     {
112       heapify(arr, n, (size_t)i, order, compare, &stats);
113     }
114
115     //           ""
116     for (size_t i = n - 1; i > 0; i--)
117     {
118       swap_elements(arr, 0, i, &stats);
119       heapify(arr, i, 0, order, compare, &stats);
120     }
121
122     if (order == ASCENDING)
123     {
124       printf("\n\033[32 m\033[0m Sorting order: \033[32masc\033[0m\n");
125     }
126     else
127     {
128       printf("\n\033[32 m\033[0m Sorting order: \033[32mdesc\033[0m\n");
129     }
130
131     printf("\033[32 m\033[0m Number of swaps: \033[32m%zu\033[0m\n", stats.
          swaps);
132     printf("\033[32 m\033[0m Number of comparisons: \033[32m%zu\033[0m\n",
          stats.comparisons);
133 }
```

2.2.4 Реалізація програми лабораторної роботи

У програмі створюється динамічний масив та заповнюється випалковими цілими числами. Також додатково реалізовані функції рахування часу виконання алгоритму та виводу масиву у консоль у приємному вигляді.

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

#include "general_utils.h"

#include "heap_sort.h"
#include "radix_sort.h"
#include "merge_sort.h"

#include "time.h"

int int_compare(const void *a, const void *b)
{
  int x = *(const int *)a;
  int y = *(const int *)b;
  return (x - y);
}

void print_array(DynArr *arr)
{
  for (int i = 0; i < arr->size; i++)
  {
    int *value = (int *)get_elem_dyn_arr(arr, i);
    if (i == 0)
    {
      printf("[ %d ", *value);
    }
    else if (i == arr->size - 1)
    {
      printf(" %d ]", *value);
    }
    else
    {
      printf(" %d ", *value);
    }
  }

  printf("\nArray size: %zu\n", arr->size);
}

void generate_arr_data(DynArr *arr)
{
  for (int i = 0; i < arr->capacity; i++)
  {
    int value = generateRandomInt(1, 99);
    push_dyn_arr(arr, &value);
  }
}

void measure_execution_time_for_heap(int (*func)(DynArr *arr, SortOrder *order, int (*compare)(const void *, const void *)), const DynArr *arr, SortOrder *order, int (*compare)(const void *, const void *))
{
```

```
52    clock_t start = clock();
53    func(arr, order, compare);
54    clock_t end = clock();
55
56    double time_taken_ms = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;
57    printf("\033[32 m\033[0m Estimated time: \033[32m%.2f\033[0m ms\n",
          time_taken_ms);
58  }
59
60  void measure_execution_time(int (*func)(DynArr *arr, SortOrder *order),
        const DynArr *arr, SortOrder *order)
61  {
62    clock_t start = clock();
63    func(arr, order);
64    clock_t end = clock();
65
66    double time_taken_ms = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;
67    printf("\033[32 m\033[0m Estimated time: \033[32m%.2f\033[0m ms\n",
          time_taken_ms);
68  }
69
70  void task1()
71  {
72    srand(time(NULL));
73
74    int ARR_SIZE = 50;
75    DynArr *int_arr = init_dyn_arr(sizeof(int), ARR_SIZE);
76
77    generate_arr_data(int_arr);
78
79    highlightText("DYNAMIC ARRAY OF INT VALUES", "blue");
80    print_array(int_arr);
81
82    highlightText("\nDYNAMIC ARRAY OF INT VALUES AFTER HEAP SORT", "yellow")
          ;
83    measure_execution_time_for_heap(heap_sort_dyn_arr, int_arr, ASCENDING,
          int_compare);
84
85    highlightText("\nDYNAMIC ARRAY OF INT VALUES AFTER RADIX SORT", "yellow"
          );
86    measure_execution_time(radix_sort_dyn_arr, int_arr, ASCENDING);
87
88    highlightText("\nDYNAMIC ARRAY OF INT VALUES AFTER MERGE SORT", "yellow"
          );
89    measure_execution_time(merge_sort_dyn_arr, int_arr, ASCENDING);
90
91    print_array(int_arr);
92
93    destroy_dyn_arr(int_arr);
94
95    return 0;
96  }
```

2.3 Результати роботи програми:

2.3.1 Тестування алгоритмів для масиву з 15 елементів

```
DYNAMIC ARRAY OF INT VALUES
[ 61 | 68 | 22 | 42 | 8 | 85 | 24 | 79 | 85 | 85 | 31 | 82 | 11 | 59 | 65 ]
Array size: 15

DYNAMIC ARRAY OF INT VALUES AFTER HEAP SORT

▲ Sorting order: asc
⇄ Number of swaps: 48
⇅ Number of comparisons: 76
⏲ Estimated time: 0.09 ms
[ 8 | 11 | 22 | 24 | 31 | 42 | 59 | 61 | 65 | 68 | 79 | 82 | 85 | 85 | 85 ]
Array size: 15
```

Рис. 1. Результат сортування частково впорядкованим деревом

```
DYNAMIC ARRAY OF INT VALUES
[ 38 | 54 | 13 | 80 | 97 | 33 | 28 | 77 | 99 | 69 | 6 | 9 | 23 | 15 | 8 ]
Array size: 15

DYNAMIC ARRAY OF INT VALUES AFTER MERGE SORT

▲ Sorting order: asc
⇄ Number of swaps: 59
⇅ Number of comparisons: 42
⏲ Estimated time: 0.06 ms
[ 6 | 8 | 9 | 13 | 15 | 23 | 28 | 33 | 38 | 54 | 69 | 77 | 80 | 97 | 99 ]
Array size: 15
```

Рис. 2. Результат сортування злиттям

```
DYNAMIC ARRAY OF INT VALUES
[ 80 | 83 | 99 | 27 | 70 | 93 | 35 | 26 | 11 | 64 | 98 | 35 | 3 | 6 | 95 ]
Array size: 15

DYNAMIC ARRAY OF INT VALUES AFTER RADIX SORT

▲ Sorting order: asc
⇄ Number of swaps: 30
⇅ Number of comparisons: 30
⏲ Estimated time: 0.06 ms
[ 3 | 6 | 11 | 26 | 27 | 35 | 35 | 64 | 70 | 80 | 83 | 93 | 95 | 98 | 99 ]
Array size: 15
```

Рис. 3. Результат порозрядного цифрового сортування

2.3.2 Аналіз результатів сортування при різних розмірах масиву

Таблиця 1. Результати сортування злиттям

Відсортований набір даних	**20**	**1000**	**5000**	**10000**	**50000**
Кількість пересилань	88	9976	61808	133616	784464
Кількість порівнянь	66	8735	55153	120130	716680
Час сортування (мс.)	0.07	0.46	1.97	4.19	21.55
Відсортований у зворотньму порядку	**20**	**1000**	**5000**	**10000**	**50000**
Кількість пересилань	88	9976	61808	133616	784464
Кількість порівнянь	66	8691	55175	120235	716606
Час сортування (мс.)	0.07	0.44	2.27	4.40	20

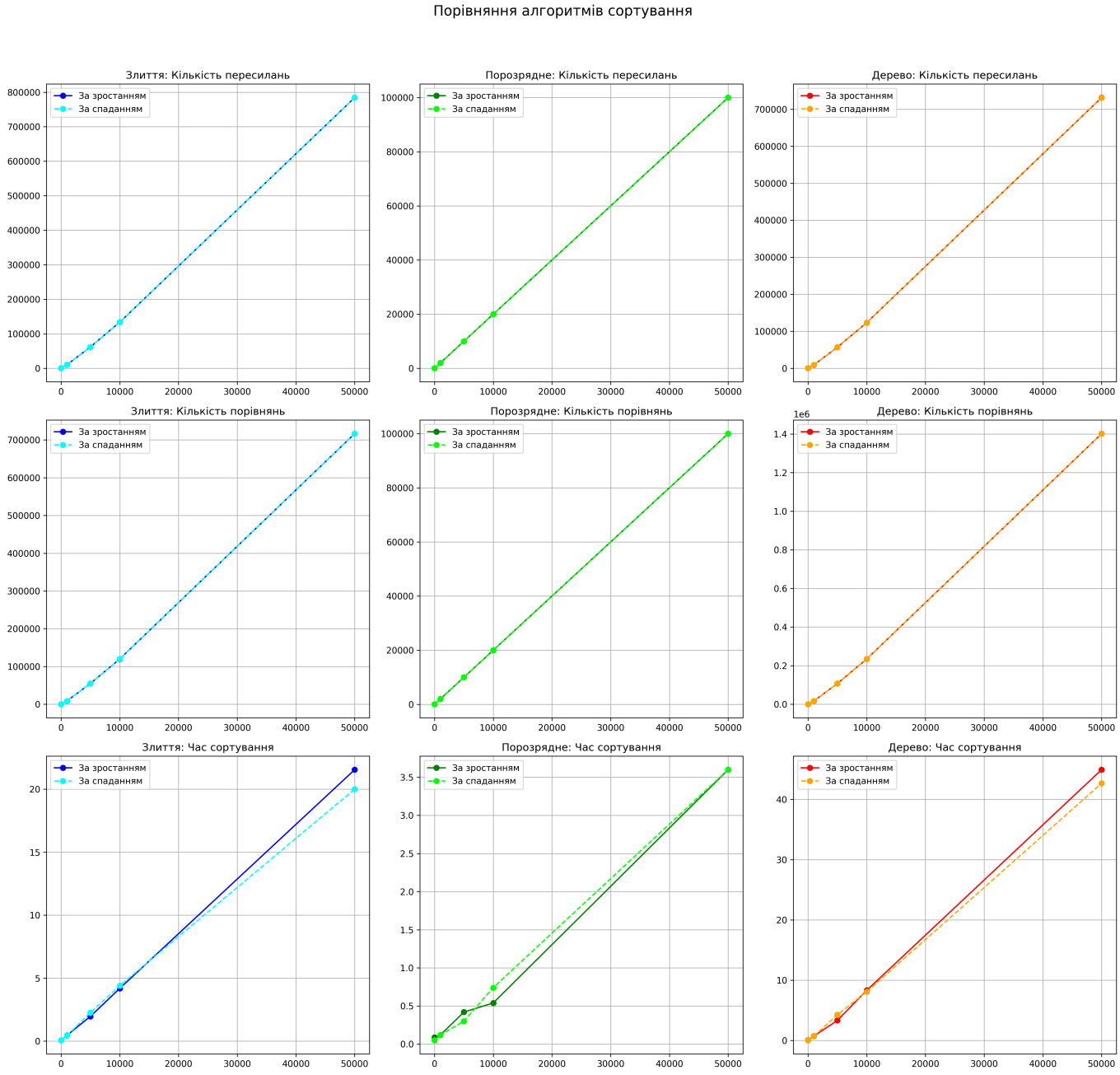
Таблиця 2. Результат порозрядного цифрового сортування

Відсортований набір даних (asc)	**20**	**1000**	**5000**	**10000**	**50000**
Кількість пересилань	40	2000	10000	20000	100000
Кількість порівнянь	40	2000	10000	20000	100000
Час сортування (мс.)	0.09	0.12	0.42	0.54	3.6
Відсортований у зворотньму порядку (desc)	**20**	**1000**	**5000**	**10000**	**50000**
Кількість пересилань	40	2000	10000	20000	100000
Кількість порівнянь	40	2000	10000	20000	100000
Час сортування (мс.)	0.05	0.12	0.3	0.74	3.6

Таблиця 3. Результат сортування випадково впорядкованим деревом

Відсортований набір даних (asc)	**20**	**1000**	**5000**	**10000**	**50000**
Кількість пересилань	74	9002	56784	123327	731544
Кількість порівнянь	113	16804	107420	234426	1401449
Час сортування (мс.)	0.07	0.74	3.33	8.33	44.95
Відсортований у зворотньму порядку (desc)	**20**	**1000**	**5000**	**10000**	**50000**
Кількість пересилань	68	9007	56637	123308	731255
Кількість порівнянь	115	16791	107248	234358	1401659
Час сортування (мс.)	0.04	0.72	4.28	8.10	42.68

Графіки побудовані на основі таблиць:



3 Висновки

В ході виконання лабораторної роботи було розроблено три алгоритми сортування відповідно варіанту завдання.

Після аналізу отриманих результатів для різних алгоритмів сортувань, можна зазначити що результати в залежності від зміни напрямку сортування (ask, desk) не дуже сильно змінюються для всіх сортувань.

Тим часом результати сортування за часом для різних напрямків сортування трішки відрізняються. Десь сортування за зростанням швидше, а десь навпаки.