МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №4

по дисциплине: Алгоритмы и структуры данных
тема: Сравнительный анализ алгоритмов поиска (Pascal/C)

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили: асс. Солонченко Роман
Евгеньевич

Белгород 2023 г.

Лабораторная работа №4

Сравнительный анализ алгоритмов поиска (Pascal/C)

Цель работы: изучение алгоритмов поиска элемента в массиве и закрепление навыков в проведении сравнительного анализа алгоритмов.

1. Листинг программы. *main.c*

```c
#include <time.h>

#include <algc.h>

#define LOW 50
#define HIGH 450
#define STEP 50

int main() {
    srand(time(0));

    comparesSearchExperiment       (linearSearch,             "linear search",
    ↪  LOW, HIGH, STEP);
    comparesSearchExperiment       (linearQuickSearch,        "quick linear search",
    ↪  LOW, HIGH, STEP);
    comparesOrderedSearchExperiment(orderedLinearQuickSearch, "quick linear search for
    ↪  ordered arrays", LOW, HIGH, STEP);
    comparesOrderedSearchExperiment(orderedBinarySearch,      "binary search",
    ↪  LOW, HIGH, STEP);
    comparesOrderedSearchExperiment(orderedBlockSearch,       "block search",
    ↪  LOW, HIGH, STEP);

    return 0;
}
```

search.h

```c
#ifndef SEARCH
#define SEARCH

#include <lab3/sorts.h>

#include <stdbool.h>

#define SEARCH_UTILITY_EXPERIMENT_ITERATIONS_AMOUNT 10000
```

```c
typedef int (*SearchingFunction)(int*, int, int, int*);
typedef int (*OrderedSearchingFunction)(int*, int, int, int*);

int linearSearch(int* a, int size,  int searchElement, int* comps);
int linearQuickSearch(int* a, int size,  int searchElement, int* comps);
int orderedLinearQuickSearch(int* a, int size,  int searchElement, int* comps);
int orderedBinarySearch(int* a, int size,  int searchElement, int* comps);
int orderedBlockSearch(int* a, int size,  int searchElement, int* comps);

void comparesSearchExperiment(SearchingFunction function, char *searchingFunctionName, int
↪  low, int high, int step);
void comparesOrderedSearchExperiment(SearchingFunction function, char
↪  *searchingFunctionName, int low, int high, int step);

#endif
```

utility.c

```c
#include <lab4/search.h>

#include <assert.h>
#include <limits.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>

void comparesSearchExperiment(SearchingFunction function, char *searchingFunctionName, int
↪  low, int high, int step) {
    printf("=============================================\n\n");
    printf("Launching comparing experiment for function %s\n\n", searchingFunctionName);

    for (int i = low; i <= high; i += step) {
        int sumCompares = 0;
        int maxCompares = INT_MIN;
        for (int j = 0; j < SEARCH_UTILITY_EXPERIMENT_ITERATIONS_AMOUNT; j++) {
            int* array = malloc((i + 1) * sizeof(int));
            genRandom(array, i);
            int compares = 0;
            int randomIndex = rand() % i;
            int foundIndex = function(array, i, array[randomIndex], &compares);
            sumCompares += compares;
            maxCompares = compares > maxCompares ? compares : maxCompares;
            assert(array[foundIndex] == array[randomIndex]);
```

```
            free(array);
        }

        int avgCompares = sumCompares / SEARCH_UTILITY_EXPERIMENT_ITERATIONS_AMOUNT;
        printf("For %3d elements. Average: %7d compares; maximum: %7d compares\n", i,
        ↪  avgCompares, maxCompares);
    }

    printf("\n");
}

void comparesOrderedSearchExperiment(SearchingFunction function, char
↪  *searchingFunctionName, int low, int high, int step) {
    printf("================================================\n\n");
    printf("Launching comparing experiment for function %s\n\n", searchingFunctionName);

    for (int i = low; i <= high; i += step) {
        int sumCompares = 0;
        int maxCompares = INT_MIN;
        for (int j = 0; j < SEARCH_UTILITY_EXPERIMENT_ITERATIONS_AMOUNT; j++) {
            int* array = malloc((i + 1) * sizeof(int));
            genOrdered(array, i);
            int compares = 0;
            int randomIndex = rand() % i;
            int foundIndex = function(array, i, array[randomIndex], &compares);
            sumCompares += compares;
            maxCompares = compares > maxCompares ? compares : maxCompares;
            assert(array[foundIndex] == array[randomIndex]);
            free(array);
        }

        int avgCompares = sumCompares / SEARCH_UTILITY_EXPERIMENT_ITERATIONS_AMOUNT;
        printf("For %3d elements. Average: %7d compares; maximum: %7d compares\n", i,
        ↪  avgCompares, maxCompares);
    }

    printf("\n");
}
```

linearsearch.c

```
#include <lab4/search.h>
```

```c
int linearSearch(int* a, int size,  int searchElement, int* comps) {
    int i = 0;
    while (INC_COMPARES(comps) && i < size && INC_COMPARES(comps) && a[i] !=
    ↪  searchElement)
        i++;

    return i;
}
```

linearquicksearch.c

```c
#include <lab4/search.h>

int linearQuickSearch(int* a, int size,  int searchElement, int* comps) {
    int i = 0;
    a[size] = searchElement;
    while (INC_COMPARES(comps) && a[i] != searchElement)
        i++;

    return i;
}
```

orderedlinearquicksearch.c

```c
#include <lab4/search.h>

int orderedLinearQuickSearch(int* a, int size,  int searchElement, int* comps) {
    int i = 0;
    while (INC_COMPARES(comps) && i < size && INC_COMPARES(comps) && a[i] !=
    ↪  searchElement) {
        if (INC_COMPARES(comps) && a[i] > searchElement)
           i = size - 1;

        i++;
    }

    return i;
}
```

binarysearch.c

```c
#include <lab4/search.h>

int orderedBinarySearch(int* a, int size, int searchElement, int* comps) {
    int left = 0, right = size - 1;

    while (INC_COMPARES(comps) && left <= right) {
        int middle = left + (right - left) / 2;

        if (INC_COMPARES(comps) && a[middle] < searchElement)
            left = middle + 1;
        else if (INC_COMPARES(comps) && a[middle] > searchElement)
            right = middle - 1;
        else
            return middle;
    }

    return size;
}
```

blocksearch.c

```c
#include <lab4/search.h>

#include <math.h>

int orderedBlockSearch(int* a, int size, int searchElement, int* comps) {
    int blockSize = sqrtl(size);
    int blocksAmount = size / blockSize + size % blockSize;
    for (int i = 0; INC_COMPARES(comps) && i < blocksAmount; i++) {
        int blockBeginIndex = i * blockSize;
        int blockEndIndex = (i + 1) * blockSize;
        blockEndIndex = INC_COMPARES(comps) && blockEndIndex > size ? size :
        ↪  blockEndIndex;

        if (INC_COMPARES(comps) && a[blockEndIndex - 1] < searchElement) continue;

        int j = blockBeginIndex;
        while (INC_COMPARES(comps) && j < blockEndIndex && INC_COMPARES(comps) && a[j] !=
        ↪  searchElement) {
            if (INC_COMPARES(comps) && a[j] > searchElement)
                j = size - 1;

            j++;
```

```
        }

        return j;
    }

    return size;
}
```

2. Результаты работы программы.

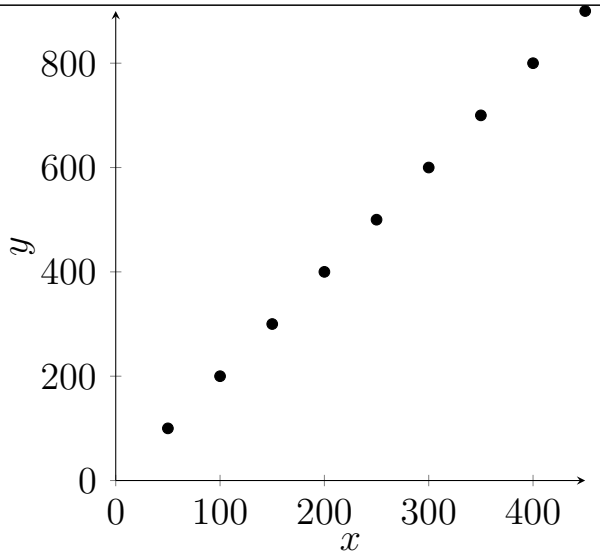
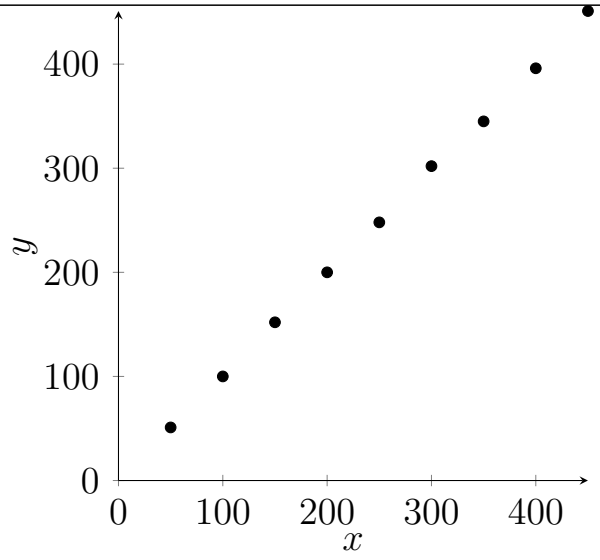
Максимальное количество операций сравнения

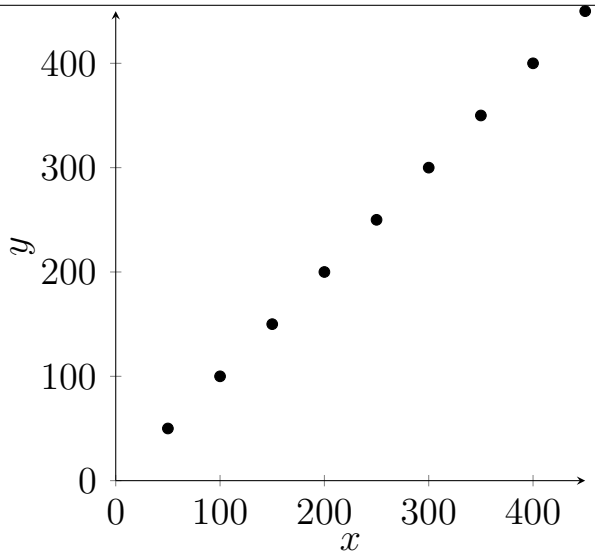
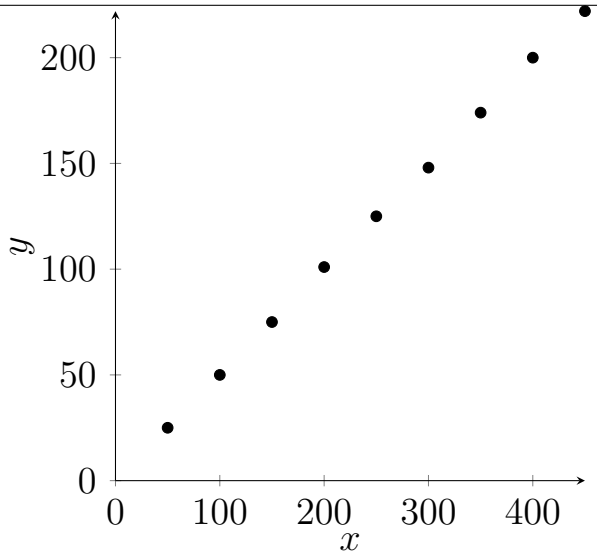
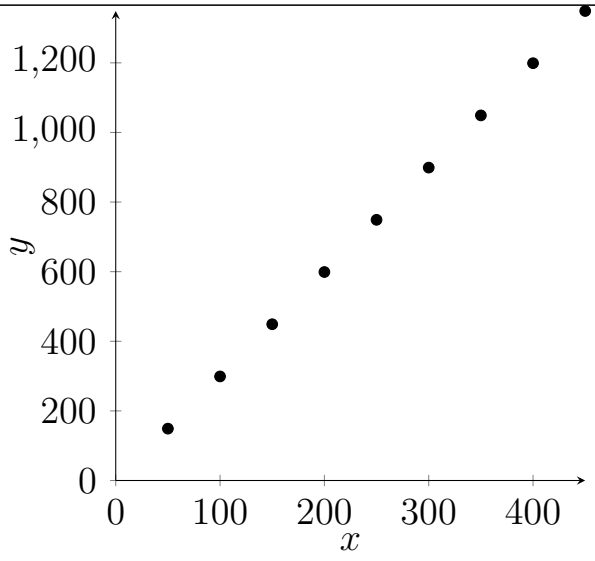
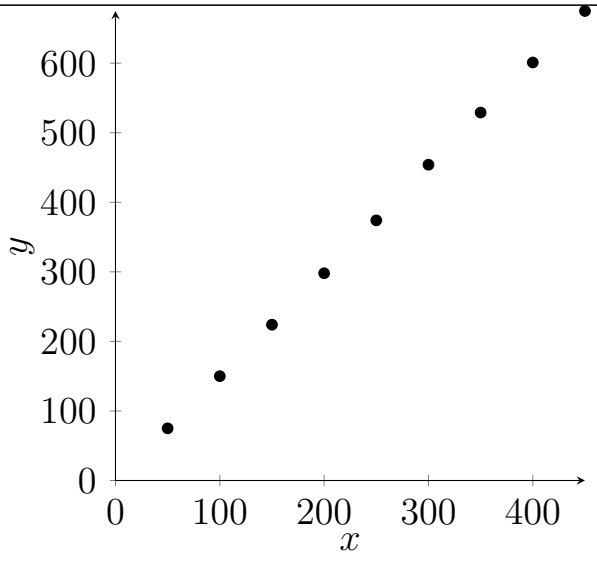
Алгоритмы поиска	Количество элементов в массиве								
	50	100	150	200	250	300	350	400	450
Линейный (неупорядоченный массив)	100	200	300	400	500	600	700	800	900
Быстрый линейный (неупорядоченный массив)	50	100	150	200	250	300	350	400	450
Быстрый линейный (упорядоченный массив)	149	299	449	599	749	899	1049	1199	1349
Бинарный (упорядоченный масив)	17	20	21	23	23	24	25	26	26
Блочный (упорядоченный масив)	41	59	71	83	92	101	110	119	125

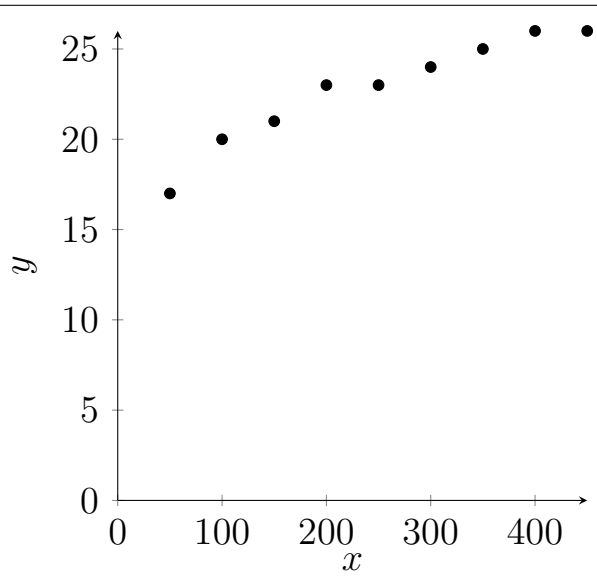
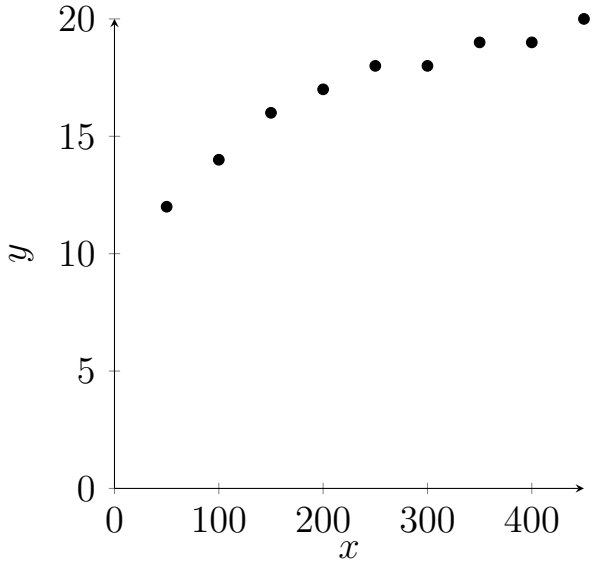
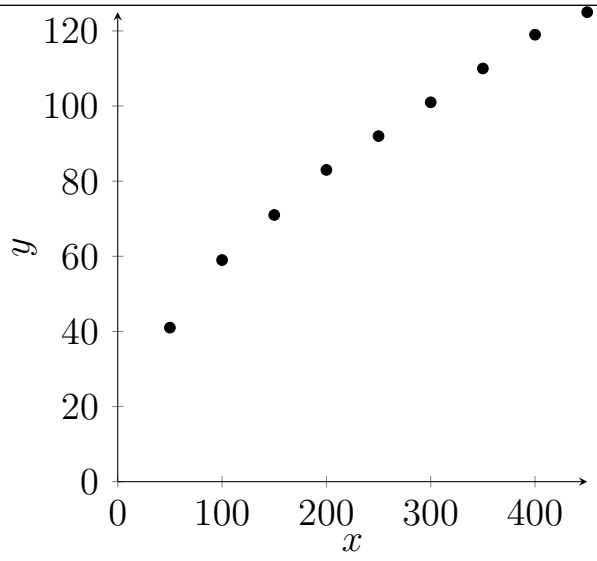
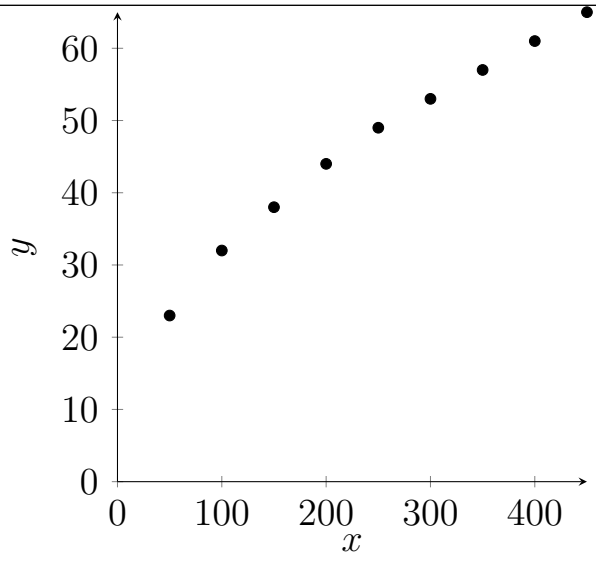
Среднее количество операций сравнения

Алгоритмы поиска	Количество элементов в массиве								
	50	100	150	200	250	300	350	400	450
Линейный (неупорядоченный массив)	51	100	152	200	248	302	345	396	451
Быстрый линейный (неупорядоченный массив)	25	50	75	101	125	148	174	200	222
Быстрый линейный (упорядоченный массив)	75	150	224	298	374	454	529	601	675
Бинарный (упорядоченный масив)	12	14	16	17	18	18	19	19	20
Блочный (упорядоченный масив)	23	32	38	44	49	53	57	61	65

3. Графики зависимостей ФВС.

Поиск	Максимальное количество операций сравнения	Среднее количество операций сравнения
Линейный поиск		

Быстрый линейный поиск (неупорядоченный массив)		
Быстрый линейный поиск (упорядоченный массив)		

Бинарный поиск (упорядоченный массив)		
Блочный поиск (упорядоченный массив)		

4. Выводы по работе.
   Линейный поиск (неупорядоченный массив):
   $t = 2 + 3 \cdot N$

   Если искомый элемент в начале массива: $\Omega(1)$
   Если искомый элемент в конце массива или искомого элемента нет в массиве: $O(N)$
   В общем случае: $\Theta(N)$
   Быстрый линейный поиск (неупорядоченный массив):
   $t = 3 + 2 \cdot N$

   Если искомый элемент в начале массива: $\Omega(1)$
   Если искомый элемент в конце массива или искомого элемента нет в массиве: $O(N)$

В общем случае: $\Theta(N)$

Быстрый линейный поиск (упорядоченный массив):

$t = 2 + 5 \cdot N$ Если искомый элемент в начале массива: $\Omega(1)$

Все элементы массива меньше искомого: $O(N)$

В общем случае: $\Theta(N)$

Бинарный поиск:

$t = 3 + 5 \cdot \log_2 N$ Если искомый элемент в середине массива: $\Omega(1)$

Если элемента нет в массиве: $O(\log_2 N)$

В общем случае: $\Theta(\log_2 N)$

Блочный поиск:

$t = 2 + 7 \cdot \sqrt{N}$ Если искомый элемент в начале массива: $\Omega(1)$

Если искомый элемент в конце массива: $O(\sqrt{N})$

В общем случае: $O(\sqrt{N})$

Ссылка на репозиторий

Вывод: в ходе лабораторной работы изучили алгоритмы поиска элемента в массиве и закрепили навыки в проведении сравнительного анализа алгоритмов.