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 C»

Выполнил: ст. группы ПВ-223

Игнатьев Артур Олегович

Проверил:

асс. Солонченко Роман Евгеньевич

Белгород 2023г.

**Лабораторная работа №4**

**«Сравнительный анализ алгоритмов поиска C»**

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

- Тема лабораторной работы;

- Цель лабораторной работы;

- Условия задач и их решение;

- Вывод.**Задание к лабораторной работе :**

1. Изучить алгоритмы поиска:

1) в неупорядоченном массиве:

- линейный;

- быстрый линейный;

2) в упорядоченном массиве:

- быстрый линейный;

- бинарный;

- блочный.

2. Разработать и программно реализовать средство для проведения экспериментов по определению временных характеристик алгоритмов поиска.

3. Провести эксперименты по определению временных характеристик алгоритмов поиска. Результаты экспериментов представить в виде таблиц 12 и 13. Клетки таблицы 12 содержат максимальное количество операций сравнения при выполнении алгоритма поиска, а клетки таблицы 13 —среднее число операций сравнения.

4. Построить графики зависимости количества операций сравнения от количества элементов в массиве.

5. Определить аналитическое выражение функции зависимости количества операций сравнения от количества элементов в массиве.

6. Определить порядок функций временной сложности алгоритмов поиска.

Листинг программы:

Файл standard\_functions.h

// Функция для обмена двух элементов массива  
void swap(void \*a, void \*b, int size);  
  
// Функция генерации рандомного массива размера size  
void generateRandomArray(int \*array, const size\_t size);  
  
// Возвращает 'истину', если массив отсортирован, иначе -- 'ложь'  
bool isOrdered(int \*array, size\_t size);  
  
// Выводит массив array размера size  
void outputArray(int \*array, size\_t size);

Файл standard\_functions.c

#include "standart\_functions.h"  
  
void swap(void \*a, void \*b, int size) {  
 char \*pa = a;  
 char \*pb = b;  
 for (int i = 0; i < size; i++, pa++, pb++) {  
 char t = \*pa;  
 \*pa = \*pb;  
 \*pb = t;  
 }  
}  
  
void generateRandomArray(int \*array, const size\_t size) {  
 srand(time(0));  
  
 for (size\_t i = 0; i < size; i++)  
 array[i] = rand() % 10000;  
}  
  
bool isOrdered(int \*array, size\_t size) {  
 for (size\_t i = 1; i < size; i++)  
 if (array[i] < array[i - 1])  
 return false;  
  
 return true;  
}  
  
  
void outputArray(int \*array, size\_t size) {  
 printf("[");  
  
 for (size\_t i = 0; i < size; i++) {  
 printf("%d", array[i]);  
  
 if (i < size - 1)  
 printf(", ");  
 }  
  
 printf("]\n");  
}

Файл sort.h

// Сортировка включением  
long long insertionSort(int A[], int n);  
  
// Сортировка выбором  
long long selectionSort(int A[], int n);  
  
// Сортировка обменом  
long long bubbleSort(int A[], int n);  
  
// Улучшенная сортировка обменом 1  
long long bubbleSort1(int arr[], int n);  
  
// Улучшенная сортировка обменом 2  
long long bubbleSort2(int arr[], int n);  
  
// Сортировка массива методом Шелла  
long long shellSort(int arr[], int n);  
  
// Сортировка Хоара (быстрая сортировка)  
long long hoarSort(int arr[], int high);  
  
// Пирамидальная сортировка  
long long heapSort(int A[], int n);  
  
// Компаратор для qsort  
int compareQsort(const void \*a, const void \*b);

Файл sort.c

#include "sort.h"  
  
long long insertionSort(int A[], int n) {  
 long long comparisons = 0;  
 int i, j, k;  
 for (j = 1; j < n; j++) {  
 k = A[j];  
 i = j - 1;  
 while (k < A[i] && i >= 0) {  
 comparisons++;  
 A[i + 1] = A[i];  
 i -= 1;  
 }  
 comparisons++;  
 A[i + 1] = k;  
 }  
 return comparisons + (n - 1);  
}  
  
long long selectionSort(int A[], int n) {  
 long long comparisons = 0;  
 int i, j, x, k;  
 for (i = 0; i < n - 1; i++) {  
 x = A[i];  
 k = i;  
 for (j = i + 1; j < n; j++)  
 if (A[j] < x) {  
 k = j;  
 x = A[k];  
 }  
 comparisons += (n - (i + 1));  
 A[k] = A[i];  
 A[i] = x;  
 }  
 return comparisons + (n - 1);  
}  
  
long long bubbleSort(int A[], int n) {  
 long long comparisons = 0;  
 int i, j, k, p;  
 for (i = 0; i < n - 1; i++) {  
 p = 0;  
 for (j = n - 1; j > i; j--) {  
 comparisons++;  
 if (A[j] < A[j - 1]) {  
 k = A[j];  
 A[j] = A[j - 1];  
 A[j - 1] = k;  
 p = 1;  
 }  
 }  
 comparisons += (n - i);  
 //Если перестановок не было, то сортировка выполнена  
 if (!p)  
 break;  
 }  
 return comparisons + (n - 1);  
}  
  
long long bubbleSort1(int arr[], int n) {  
 long long comparisons = 0;  
 int temp;  
 bool swapped;  
 for (int i = 0; i < n - 1; i++) {  
 swapped = false;  
 for (int j = 0; j < n - i - 1; j++) {  
 comparisons++;  
 if (arr[j] > arr[j + 1]) {  
 // меняем элементы местами  
 temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 swapped = true;  
 }  
 }  
 comparisons += (n - i);  
 // если на текущей итерации не было ни одного обмена,  
 // то массив уже отсортирован и можно завершить процесс  
 if (swapped == false)  
 break;  
 }  
 return comparisons + (n - 1);  
}  
  
long long bubbleSort2(int arr[], int n) {  
 long long comparisons = 0;  
 int i, j, temp;  
 int lastSwapIndex = n - 1;  
 for (int i = 0; i < n - 1; i++) {  
 int currentSwapIndex = -1;  
 for (int j = 0; j < lastSwapIndex; j++) {  
 comparisons++;  
 if (arr[j] > arr[j + 1]) {  
 // меняем элементы местами  
 temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 currentSwapIndex = j;  
 }  
 }  
 comparisons += (lastSwapIndex + 1);  
 // если на текущей итерации не было ни одного обмена,  
 // то массив уже отсортирован и можно завершить процесс  
 if (currentSwapIndex == -1)  
 break;  
 lastSwapIndex = currentSwapIndex;  
 }  
 return comparisons + (n - 1);  
}  
  
long long shellSort(int arr[], int n) {  
 long long comparisons = 0;  
 // Начинаем с большего шага  
 for (int gap = n / 2; gap > 0; gap /= 2) {  
 comparisons++;  
 // Проходим по элементам массива с шагом gap  
 for (int i = gap; i < n; i++) {  
 // Сохраняем текущий элемент в переменную temp  
 int temp = arr[i];  
 // Сдвигаем предыдущие элементы, которые больше текущего, на один шаг вперед  
 int j;  
 for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {  
 comparisons++;  
 arr[j] = arr[j - gap];  
 }  
 comparisons++;  
 // Вставляем текущий элемент на правильную позицию  
 arr[j] = temp;  
 }  
 comparisons += (n - gap);  
 }  
 return comparisons + 1;  
}  
  
//Эта функция принимает последний элемент в качестве опорного, помещает  
//этот элемент в правильное положение в отсортированном массиве и помещает  
//все меньшие (меньше опорного) элементы слева от него и все большие  
//элементы справа от него  
int partition(int arr[], int low, int high) {  
 int support = arr[high]; // опорный элемент  
 int i = (low - 1); // индекс меньшего элемента  
 for (int j = low; j <= high - 1; j++) {  
 // Если текущий элемент меньше или равен опорному  
 if (arr[j] <= support) {  
 i++; // увеличиваем индекс меньшего элемента  
 swap(&arr[i], &arr[j], sizeof(arr[i]));  
 }  
 }  
 swap(&arr[i + 1], &arr[high], sizeof(arr[i + 1]));  
 return (i + 1);  
}  
  
// Функция для реализации алгоритма быстрой сортировки  
//arr[] - Массив для сортировки,  
//low - Начальный индекс,  
//high - Конечный индекс  
long long q\_sort(int arr[], int low, int high, long long comparisons) {  
 if (low < high) {  
 // separative - это разделительный индекс, arr[sep] сейчас на правильном месте  
 int separative = partition(arr, low, high);  
 comparisons += (2 \* (high - low));  
 // Рекурсивно сортируем элементы до разделителя и после разделителя  
 q\_sort(arr, low, separative - 1, comparisons);  
 q\_sort(arr, separative + 1, high, comparisons);  
 }  
 return comparisons + 1;  
}  
  
long long hoarSort(int arr[], int high) {  
 return q\_sort(arr, 0, high, 0);  
}  
  
void sift(int A[], int L, int R) {  
 int i, j, x, k;  
 i = L;  
 j = 2 \* L + 1;  
 x = A[L];  
 if ((j < R) && (A[j] < A[j + 1]))  
 j++;  
 while ((j <= R) && (x < A[j])) {  
 k = A[i];  
 A[i] = A[j];  
 A[j] = k;  
 i = j;  
 j = 2 \* j + 1;  
 if ((j < R) && (A[j] < A[j + 1]))  
 j++;  
 }  
}  
  
long long heapSort(int A[], int n) {  
 long long comparisons = 0;  
 int L, R, x, i;  
 L = n / 2;  
 R = n - 1;  
 // Построение пирамиды из исходного массива  
 while (L > 0) {  
 comparisons++;  
 L = L - 1;  
 sift(A, L, R);  
 }  
 comparisons++;  
 // Сортировка: пирамида в отсортированный массив  
 while (R > 0) {  
 comparisons++;  
 x = A[0];  
 A[0] = A[R];  
 A[R] = x;  
 R--;  
 sift(A, L, R);  
 }  
 return comparisons + 1;  
}  
  
int compareQsort(const void \*a, const void \*b) {  
 int arg1 = \*(const int \*) a;  
 int arg2 = \*(const int \*) b;  
 if (arg1 < arg2)  
 return -1;  
 if (arg1 > arg2)  
 return 1;  
 return 0;  
}

Файл lab3.h

#define ARRAY\_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))  
  
typedef struct sortFunction {  
 long long (\*sort)(int[], int);  
 char \*name;  
} sortFunction;  
  
typedef struct generationFunction {  
 void (\*generate)(int \*, size\_t);  
 char \*name;  
} generationFunction;  
  
void timeExperiment();

Файл search.h

// Линейный поиск элемента x в массиве arr размера size  
long long linearSearch(const long long const \*arr, const size\_t size, const long long x);  
  
// Быстрый линейный поиск элемента x в массиве arr размера size  
long long fastLinearSearch(long long \*arr, const size\_t size, const long long x);  
  
// Быстрый линейный поиск элемента x в отсортированном массиве arr размера size. Массив должен  
// быть упорядочен  
long long fastLinearSearchSortedArray(long long \*arr, const size\_t size, const long long x);  
  
// Бинарный поиск элемента x в подмассиве массива arr от элемента arr[left] до элемента arr[right].  
// Подмассив должен быть упорядочен  
long long binarySearchSubarray(long long \*arr, long long left, long long right,  
 const long long x);  
  
// Бинарный поиск элемента x в массиве arr размера size. Массив должен быть упорядочен  
long long binarySearch(long long \*arr, const size\_t size, const long long x);  
  
// Блочный поиск элемента x в массиве arr размера size. Массив должен быть отсортирован  
long long blockSearch(long long \*arr, const size\_t size, const long long x);

Файл search.c

long long linearSearch(const long long const \*arr, const size\_t size, const long long x) {  
 for (long long i = 0; i < size; i++)  
 if (arr[i] == x)  
 return i;  
 return -1;  
}  
  
long long fastLinearSearch(long long \*arr, const size\_t size, const long long x) {  
 arr[size] = x;  
 long long i = 0;  
 while (arr[i] != x)  
 i++;  
 return i != size ? i : -1;  
}  
  
long long fastLinearSearchSortedArray(long long \*arr, const size\_t size, const long long x) {  
 for (int i = 0; i < size; i++) {  
 if (arr[i] == x)  
 return i;  
 if (arr[i] > x)  
 return -1;  
 }  
 return -1;  
}  
  
long long binarySearchSubarray(long long \*arr, long long left, long long right,  
 const long long x) {  
 if (right >= left) {  
 long long mid = left + (right - left) / 2;  
 // Если элемент находится в середине  
 if (arr[mid] == x)  
 return mid;  
 // Если элемент меньше, чем mid, то он может быть только в левом подмассиве  
 if (arr[mid] > x)  
 return binarySearchSubarray(arr, left, mid - 1, x);  
 // В противном случае элемент может быть только в правом подмассиве  
 return binarySearchSubarray(arr, mid + 1, right, x);  
 }  
 // Возвращаем -1, если элемент не найден  
 return -1;  
}  
  
long long binarySearch(long long \*arr, const size\_t size, const long long x) {  
 return binarySearchSubarray(arr, 0, size - 1, x);  
}  
  
long long blockSearch(long long \*arr, const size\_t size, const long long x) {  
 // Вычисляем размер блока для поиска  
 long long blockSize = sqrt(size);  
  
 // Находим блок, в котором находится искомый элемент  
 long long i;  
 for (i = 0; i < size; i += blockSize)  
 if (arr[i] > x)  
 break;  
 if (i + blockSize >= size)  
 i += blockSize;  
 // Выполняем линейный поиск в найденном блоке  
 for (long long j = i - blockSize; j < i; ++j)  
 if (arr[j] == x)  
 return j;  
 // Возвращаем -1, если элемент не найден  
 return -1;  
}

Файл lab4.h

#define ARRAY\_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))  
  
typedef struct searchFunction {  
 long long (\*search)(long long \*, size\_t, long long);  
 char \*name;  
} searchFunction;  
  
void timeExperimentSearches();  
  
// Возвращает количество операций сравнения проведённых в функции linearSearch  
long long linearSearchExperiment(long long \*arr, const size\_t size, const long long x);  
  
// Возвращает количество операций сравнения проведённых в функции fastLinearSearch  
long long fastLinearSearchExperiment(long long \*arr, const size\_t size, const long long x);  
  
// Возвращает количество операций сравнения проведённых в  
// функции fastLinearSearchSortedArrayExperiment  
long long fastLinearSearchSortedArrayExperiment(long long \*arr, const size\_t size,  
 const long long x);  
  
// Возвращает количество операций сравнения проведённых в функции binarySearchSubArray  
long long binarySearchSubArrayExperiment(long long \*arr, long long left, long long right,  
 const long long x);  
  
// Возвращает количество операций сравнения проведённых в функции binarySearch  
long long binarySearchExperiment(long long \*arr, const size\_t size, const long long x);  
  
// Возвращает количество операций сравнения проведённых в функции blockSearch  
long long blockSearchExperiment(long long \*arr, const size\_t size, const long long x);  
  
void testLinearSearch();  
  
void testFastLinearSearch();  
  
void testFastLinearSearchSortedArray();  
  
void testBinarySearch();  
  
void testBlockSearch();

Файл lab4.c

long long linearSearchExperiment(long long \*arr, const size\_t size, const long long x) {  
 long long comparisons = 0;  
  
 for (long long i = 0; i < size; i++) {  
 comparisons += 2;  
  
 if (arr[i] == x)  
 return comparisons;  
 }  
  
 return comparisons + 1;  
}  
  
long long fastLinearSearchExperiment(long long \*arr, const size\_t size, const long long x) {  
 arr[size] = x;  
  
 long long i = 0;  
 long long comparisons = 1;  
  
 while (arr[i] != x) {  
 comparisons++;  
 i++;  
 }  
  
 return comparisons + 1;  
}  
  
long long fastLinearSearchSortedArrayExperiment(long long \*arr, const size\_t size,  
 const long long x) {  
 long long comparisons = 1;  
  
 for (int i = 0; i < size; i++) {  
 if (arr[i] == x)  
 return comparisons + 2;  
  
 if (arr[i] > x)  
 return comparisons + 3;  
  
 comparisons += 3;  
 }  
  
 return comparisons;  
}  
  
long long binarySearchSubArrayExperiment(long long \*arr, long long left, long long right,  
 const long long x) {  
 long long maxIndex = right;  
 long long comparisons = 1;  
  
 while (right - left > 1) {  
 long long middle = left + (right - left) / 2;  
  
 comparisons += 2;  
  
 if (arr[middle] > x)  
 right = middle;  
 else  
 left = middle;  
 }  
  
 return comparisons + 1;  
}  
  
long long binarySearchExperiment(long long \*arr, const size\_t size, const long long x) {  
 return binarySearchSubArrayExperiment(arr, -1, size, x);  
}  
  
long long blockSearchExperiment(long long \*arr, const size\_t size, const long long x) {  
 long long comparisons = 1;  
  
 if (arr[0] > x)  
 return comparisons;  
  
 long long block = sqrt(size);  
 long long i = 0;  
  
 comparisons++;  
  
 while (i < size) {  
 comparisons += 2;  
  
 if (arr[i] > x) {  
 break;  
 }  
  
 i += block;  
 }  
  
 return comparisons + binarySearchSubArrayExperiment(arr, i - block - 1, i, x);  
}  
  
void checkTimeSearches(long long (\*sortFunc)(long long \*, size\_t, long long),  
 void (\*generateFunc)(int \*, size\_t), size\_t size,  
 char \*experimentName, long long nameSearch) {  
 static size\_t runNum = 1;  
 static int odometer[100000000];  
  
 generateFunc(odometer, size);  
  
 // Если вызывается поиск, который требует отсортированности массива, то массив будет сортироваться  
 if (nameSearch == 2 || nameSearch == 3 || nameSearch == 4)  
 qsort(odometer, size, sizeof(int), compareQsort);  
  
 printf("Запуск #%zu | ", runNum++);  
 printf("Название: %s\n", experimentName);  
  
 long long сomparison = sortFunc(odometer, size, -1);  
  
 printf("Состояние: ");  
  
 printf("OK! Подсчётов %lld\n\n", сomparison);  
  
 char filename[256];  
  
 sprintf(filename, "data/%s.csv", experimentName);  
  
 FILE \*f = fopen(filename, "a");  
  
 if (f == NULL) {  
 printf("Ошибка открытия файла %s", filename);  
  
 exit(1);  
 }  
  
 fprintf(f, "%llu; %lld\n", size, сomparison);  
  
 fclose(f);  
}  
  
void timeExperimentSearches() {  
 searchFunction searches[] = {  
 {linearSearchExperiment, "linearSearch"},  
 {fastLinearSearchExperiment, "fastLinearSearch"},  
 {fastLinearSearchSortedArrayExperiment,  
 "fastLinearSearchSortedArrayExperiment"},  
 {binarySearchExperiment, "binarySearch"},  
 {blockSearchExperiment, "blockSearch"},  
 };  
 const unsigned FUNCS\_N = ARRAY\_SIZE(searches);  
 generationFunction generation[] = {  
 {generateRandomArray, "Random"}  
 };  
 const unsigned CASES\_N = ARRAY\_SIZE(generation);  
  
 for (size\_t size = 5; size <= 45; size += 5) {  
 printf("------------------------------------------------\n");  
 printf("Размер: %llu\n", size);  
  
 for (size\_t i = 0; i < FUNCS\_N; i++)  
 for (size\_t j = 0; j < CASES\_N; j++) {  
 static char filename[128];  
  
 sprintf(filename, "%s%sTime", searches[i].name, generation[j].name);  
  
 checkTimeSearches(searches[i].search, generation[j].generate,  
 size, filename, i);  
 }  
  
 printf("\n");  
 }  
}  
  
void testLinearSearch() {  
 printf("Линейный поиск тест...\n");  
  
 // Генерация рандомного массива из остатков от деления рандомного числа на 100  
 size\_t size = 30;  
 long long \*arr1 = (long long \*) malloc(sizeof(long long) \* size);  
 generateRandomArray(arr1, size);  
  
 // Остаток не может быть отрицательным  
 assert(linearSearch(arr1, size, -1) == -1);  
 free(arr1);  
 printf("Тест 1 OK!\n");  
  
 long long arr2[] = {88, 72, 95, 56, 1, 45, 95, 27, 6, 96,  
 95, 27, 92, 9, 66, 28, 87, 61, 40,  
 84, 76, 81, 35, 80, 49, 75, 29, 90,  
 74, 5};  
  
 assert(linearSearch(arr2, size, 1) == 4);  
 printf("Тест 2 OK!\n");  
  
 printf("\n");  
}  
  
void testFastLinearSearch() {  
 printf("Быстрый линейный поиск тест...\n");  
  
 // Генерация рандомного массива из остатков от деления рандомного числа на 100  
 size\_t size = 30;  
 long long \*arr1 = (long long \*) malloc(sizeof(long long) \* size);  
 generateRandomArray(arr1, size);  
  
 // Остаток не может быть отрицательным  
 assert(fastLinearSearch(arr1, size, -1) == -1);  
 free(arr1);  
 printf("Тест 1 OK!\n");  
  
 long long arr2[] = {88, 72, 95, 56, 1, 45, 95, 27, 6, 96,  
 95, 27, 92, 9, 66, 28, 87, 61, 40,  
 84, 76, 81, 35, 80, 49, 75, 29, 90,  
 74, 5};  
  
 assert(fastLinearSearch(arr2, size, 1) == 4);  
 printf("Тест 2 OK!\n");  
  
 printf("\n");  
}  
  
void testFastLinearSearchSortedArray() {  
 printf("Быстрый линейный поиск в отсортированном массиве тест...\n");  
  
 size\_t size = 30;  
 long long arr1[] = {1, 2, 3, 5, 7, 8, 10, 12, 14, 15,  
 16, 17, 18, 20, 21, 22, 23, 24,  
 25, 26, 27, 28, 29, 30, 31, 32,  
 33, 34, 35, 36};  
  
 assert(fastLinearSearchSortedArray(arr1, size, -1) == -1);  
  
 printf("Тест 1 OK!\n");  
  
 long long arr2[] = {1, 5, 6, 9, 27, 27, 28, 29, 35,  
 40, 45, 49, 56, 61, 66, 72, 74,  
 75, 76, 80, 81, 84, 87, 88, 90,  
 92, 95, 95, 95, 96};  
  
 assert(fastLinearSearchSortedArray(arr2, size, 1) == 0);  
 printf("Тест 2 OK!\n");  
  
 printf("\n");  
}  
  
void testBinarySearch() {  
 printf("Бинарный поиск тест...\n");  
  
 size\_t size = 30;  
 long long arr1[] = {1, 2, 3, 5, 7, 8, 10, 12, 14, 15,  
 16, 17, 18, 20, 21, 22, 23, 24,  
 25, 26, 27, 28, 29, 30, 31, 32,  
 33, 34, 35, 36};  
  
 assert(binarySearch(arr1, size, -1) == -1);  
  
 printf("Тест 1 OK!\n");  
  
 long long arr2[] = {1, 5, 6, 9, 27, 27, 28, 29, 35,  
 40, 45, 49, 56, 61, 66, 72, 74,  
 75, 76, 80, 81, 84, 87, 88, 90,  
 92, 95, 95, 95, 96};  
  
 assert(binarySearch(arr2, size, 1) == 0);  
 printf("Тест 2 OK!\n");  
  
 printf("\n");  
}  
  
void testBlockSearch() {  
 printf("Блок поиска тест...\n");  
  
 size\_t size = 30;  
 long long arr1[] = {1, 2, 3, 5, 7, 8, 10, 12, 14, 15,  
 16, 17, 18, 20, 21, 22, 23, 24,  
 25, 26, 27, 28, 29, 30, 31, 32,  
 33, 34, 35, 36};  
  
 assert(blockSearch(arr1, size, -1) == -1);  
  
 printf("Тест 1 OK!\n");  
  
 long long arr2[] = {1, 5, 6, 9, 27, 27, 28, 29, 35,  
 40, 45, 49, 56, 61, 66, 72, 74,  
 75, 76, 80, 81, 84, 87, 88, 90,  
 92, 95, 95, 95, 96};  
  
 assert(blockSearch(arr2, size, 1) == 0);  
 printf("Тест 2 OK!\n");  
  
 printf("\n");  
}

Файл main.c

int main() {  
 SetConsoleOutputCP(CP\_UTF8);  
  
 testLinearSearch();  
 testFastLinearSearch();  
 testFastLinearSearchSortedArray();  
 testBinarySearch();  
 testBlockSearch();  
 timeExperimentSearches();  
 return 0;  
}

Временные характеристики алгоритмов (при поиске элемента, которого нет в массиве (-1)):

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Поиск | Количество элементов в массиве | | | | | | | | |
| 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |
| Линейный | 11 | 12 | 22 | 22 | 32 | 32 | 42 | 42 | 52 |
| Быстрый линейный | 7 | 7 | 12 | 12 | 17 | 17 | 22 | 22 | 27 |
| Быстрый линейный в отсортированном массиве | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Бинарный | 6 | 8 | 10 | 10 | 10 | 10 | 12 | 12 | 12 |
| Блочный | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Временные характеристики алгоритмов (при поиске элемента под индексом 3)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Поиск | Количество элементов в массиве | | | | | | | | |
| 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |
| Линейный | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| Быстрый линейный | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Быстрый линейный в отсортированном массиве | 3 | 10 | 10 | 10 | 10 | 27 | 27 | 27 | 45 |
| Бинарный | 3 | 4 | 4 | 5 | 5 | 5 | 5 | 6 | 6 |
| Блочный | 2 | 4 | 4 | 5 | 5 | 6 | 6 | 6 | 7 |

Графики зависимости функций временной сложности:

Порядок функций временной сложности:

|  |  |
| --- | --- |
| Поиски | Порядок функций временной сложности |
| Линейный |  |
| Быстрый линейный |  |
| Быстрый линейный в отсортированном массиве |  |
| Бинарный |  |
| Блочный |  |

Вывод: в ходе выполнения лабораторной работы были изучены алгоритмы поиска элемента в массиве и закреплены навыки в проведении сравнительного анализа алгоритмов.