**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по практической работе №2**

**по дисциплине «Программирование»**

Тема: Одномерные статические массивы.

|  |  |  |
| --- | --- | --- |
| Студент гр. 4372 |  | Барабанов И. В. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2024

**Цель работы.**

Реализовать и сравнить различные алгоритмы сортировки и поиска в массиве целых чисел.

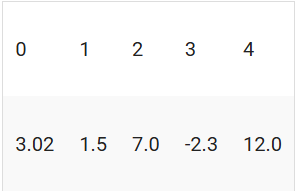
**Основные теоретические положения.**

Массив представляет собой индексированную последовательность однотипных элементов с заранее определенным количеством элементов. Наглядно одномерный массив можно представить, как набор пронумерованных ячеек, в каждой из которых содержится определенное значение.

Все массивы можно разделить на две группы: одномерные и многомерные. Описание массива в программе отличается от объявления обычной переменной наличием размерности массива, которая задается в квадратных скобках после имени.

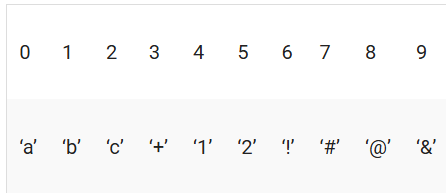
Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, const и инициализатор), что и для простых переменных.

Аналогом одномерного массива из математики может служить последовательность некоторых элементов с одним индексом: ai*ai*​ при  i = 0, 1, 2, … n – одномерный вектор. Каждый элемент такой последовательности представляет собой некоторое значение определенного типа данных. Наглядно одномерный массив можно представить как набор пронумерованных ячеек, в каждой из которых содержится определенное значение:



Это пример одномерного массива из 5 элементов, каждый из которых представляет собой некоторое вещественное значение и каждое из этих значений имеет индекс от 0 до 4.

А вот пример одномерного массива из десяти элементов, представляющих собой одиночные символы:



Каждый элемент в этих массивах определяется значением индекса элемента. Например, в последнем массиве элемент с индексом 5 равен символу  ‘2’.

Объявление в программах одномерных массивов выполняется в соответствии со следующим правилом:

**<Базовый тип элементов> <Идентификатор массива> [<Количество элементов>]**

**Значения индексов элементов массивов всегда начинается с 0**. Поэтому максимальное значение индекса элемента в массиве всегда на единицу меньше количества элементов в массиве.

Обращение к определенному элементу массива осуществляется с помощью указания значения индекса этого элемента.

При обращении к конкретному элементу массива этот элемент можно рассматривать как обычную переменную, тип которой соответствует базовому типу элементов массива, и осуществлять со значением этого элемента любые операции, которые характерны для базового типа.

Значения всех элементов массива в памяти располагаются в непрерывной области одно за другим. Общий объем памяти, выделяемый компилятором для массива, определяется как произведение объема одного элемента массива на количество элементов в массиве.

Многомерные массивы определяются аналогично одномерным массивам. Количество элементов по каждому измерению указывается отдельно в квадратных скобках:

int  A1 [5] [3];                 //  Двумерный массив, элементами которого являются                                          //   значения типа int double D [10] [15] [3];   //  Трехмерный массив, элементами которого являются                                          //   значения типа double

Общее число элементов в многомерном массиве определяется как произведение количества элементов по каждому измерению. Также как и в одномерном массиве, элементы многомерных массивов располагаются друг за другом в непрерывном участке памяти.

При определении многомерные массивы могут инициализироваться определенными значениями.

Для доступа к определенному элементу многомерного массива необходимо указать в квадратных скобках конкретные значения всех индексов этого элемента.

Простейший циклический алгоритм ввода и вывода значений элементов некоторого одномерного массива выглядит так:

const int size = 5;

int arr[size];

// Ввод элементов массива

cout << "Введите " << size << " элементов массива: ";

for (int i = 0; i < size; i++) {

cin >> arr[i];

}

// Вывод элементов массива

cout << "Элементы массива: ";

for (int i = 0; i < size; i++) {

cout << arr[i] << " ";

}

При заполнении двумерных массивов ввод значений элементов лучше осуществлять в табличной форме:

const int n = 10, m = 5;

short A [n] [m];

…

cout  << "Введите " << n << " строк из "<< m << " целых чисел: \n" for (int i = 0; i < n; ++i) {

       cout << "Строка " <<  i + 1 << ": ";        //  Вводим поэлементно значения i-й строки массива

for (int j = 0; j < m; ++j)

cin  >>  A [i] [j];        //  Очищаем поток ввода от возможных лишних введенных чисел

cin.sync();

   }

Вывод двумерных массивов, как правило, осуществляется в табличной форме. Реализация такого алгоритма может быть, например, такой:

const int n = 10, m = 10;

short A [n] [m];

for (int i = 0; i < n; ++i) //  Выводим i-ю строку массива {        for (int j = 0; j < m; ++j)        //  Выводим j-й элемент i-й строки массива

             cout << setw(7) << right << A [i] [j];

         cout << endl;

}

Сортировка – процесс размещения элементов заданного множества объектов в определенном порядке. Когда элементы отсортированы, их проще найти, производить с ними различные операции. Сортировка напрямую влияет на скорость алгоритма, в котором нужно обратиться к определенному элементу массива.

Простейшая из сортировок – сортировка обменом (пузырьковая сортировка). Вся суть метода заключается в попарном сравнении элементов и последующем обмене. Таким образом, если следующий элемент меньше текущего, то они меняются местами, максимальный элемент массива постепенно смещается в конец массива, а минимальный – в начало. Один полный проход по массиву может гарантировать, что в конце массива находится максимальный элемент.

Затем процесс повторяется до тех пор, пока вся последовательность не будет упорядочена. Важно заметить, что после первого прохода по массиву, уже имеется один упорядоченный элемент, он стоит на своем месте, и менять его не надо. Таким образом на следующем шаге будут сравниваться *N*-1 элемент.

Shaker sort – модификация пузырьковой сортировки. Принцип работы этой сортировки аналогичен bubble sort: попарное сравнение элементов и последующий обмен местами. Но имеется существенное отличие. Как только максимальный элемент становится на свое место, алгоритм не начинает новую итерацию с первого элемента, а запускает сортировку в обратную сторону. Алгоритм гарантирует, что после выполнения первой итерации, минимальный и максимальный элемент будут в начале и конце массива соответственно.

Затем процесс повторяется до тех пор, пока массив не будет отсортирован. За счет того, что сортировка работает в обе стороны, массив сортируется на порядок быстрее. Очевидным примером этого был бы случай, когда в начале массива стоит максимальный элемент, а в конце массива – минимальный. Shaker sort справится с этим за 1 итерацию, при условии, что другие элементы стоят на правильном месте.

Comb sort (сортировка расческой) – ещё одна модификация сортировки пузырьком. Алгоритм был разработан специально для случаев, когда минимальные элементы стоят слишком далеко, или максимальные – слишком близко к началу массива. В сортировке расческой переставляются элементы, стоящие на расстоянии.

Оптимально изначально взять расстояние равным длине массива , а далее уменьшать его на определенный коэффициент, который примерно равен 1.247. Когда расстояние станет равно 1, выполняется обычная сортировка пузырьком.

Сортировка вставками (insert sort) – алгоритм сортировки, в котором элементы массива просматриваются по одному, и каждый новый элемент размещается в подходящее место среди ранее упорядоченных элементов.

Общая суть сортировки вставками такова:

1)    Перебираются элементы в неотсортированной части массива.

2)    Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находится.

Сортировка вставками делить массив на 2 части – отсортированную и неотсортированную. С каждым новым элементом отсортированная часть будет увеличиваться, а неотсортированная уменьшаться. Причем найти нужное место для очередного элемента в отсортированном массиве достаточно легко.

Быстрая сортировка (quick sort) – одна из самых быстрых сортировок. Эта сортировка по сути является существенно улучшенной версией алгоритма пузырьковой сортировки.

Общая идея алгоритма состоит в том, что сначала выбирается из массива элемент, который называется опорным. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность. Затем необходимо сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: меньше опорного, раны опорному и больше опорного. Для меньших и больших значений необходимо выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

Алгоритм бинарного поиска – классический алгоритм поиска в отсортированном массиве, который использует дробление массива на половины. Если элемент, который необходимо найти, присутствует в списке, то бинарный поиск возвращает ту позицию, в которой он был найден. Бинарный поиск работает только в топ случае, если массив отсортирован. Например, если бы искомое минимальное значение стояло не на своем положенном месте, а на месте максимального элемента, то мы бы откинули его на первой же итерации. Сам алгоритм имеет вид:

1)    Определение значения в середине массива (или иной структуры данных). Полученное значение сравнивается с ключом (значением, которое необходимо найти).

2)    Если ключ меньше значения середины, то необходимо осуществлять поиск в первой половине элементов, иначе – во второй.

3)    Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.

4)    Процесс продолжается до тех пор, пока не будет определен элемент, равный значению ключа или не станет пустым интервал для поиска.

**Постановка задачи.**

Необходимо написать программу, которая:

1. Создает целочисленный массив размерности *N* = 100. Элементы массивы должны принимать случайное значение в диапазоне от -99 до 99.
2. Отсортировать заданный в пункте 1 массив […] сортировкой (от меньшего к большему). Определить время, затраченное на сортировку, используя библиотеку chrono.
3. Найти максимальный и минимальный элемент массива. Подсчитайте время поиска этих элементов в отсортированном массиве и неотсортированном, используя библиотеку chrono.
4. Выводит среднее значение (если необходимо, число нужно округлить) максимального и минимального значения в отсортированном и неотсортированном. Выводит индексы всех элементов, которые равны этому значению, и их количество. Подсчитайте время поиска.
5. Выводит количество элементов в отсортированном массиве, которые меньше числа *a*, которое инициализируется пользователем.
6. Выводит количество элементов в отсортированном массиве, которые больше числа *b*, которое инициализируется пользователем.
7. Выводит информацию о том, есть ли введенное пользователем число в отсортированном массиве. Реализуйте алгоритм бинарного поиска. Сравните скорость его работы с обычным перебором. (\*)
8. Меняет местами элементы массива, индексы которых вводит пользователь. Выведите скорость обмена, используя библиотеку chrono.

Должна присутствовать возможность запуска каждого пункта многократно.

**Выполнение работы.**

| Ввод пользователем и обработка данных | Работа алгоритма и вывод на экран |
| --- | --- |
| Создание случайного целочисленного массива | |
| При запуске программы пользователю отображается сгенерированный массив и его сортировка | Генерация случайного массива чисел и сортировка алгоритмом Quicksort:    Все дальнейшие подзадачи пользователь может выполнять в произвольном порядке, указывая номер подзадачи от 1 до 9 включительно. Выход из программы определён вводом цифры 0. |
| Сравнение алгоритмов сортировки | |
| Пользователь вводит цифру «2» и наблюдает расчёт времени сортировки массива разными алгоритмами | Выполняется сортировка сгенерированного массива разными алгоритмами сортировки, затраченное время выводится на экран: |
| Поиск минимального и максимального элементов массива | |
| Пользователь вводит цифру «3» и получает информацию о минимальном и максимальном элементах массива. | Программа вычисляет минимальный и максимальный элементы отсортированного и не отсортированного массивов: |
| Среднее значение минимального и максимального элементов | |
| Пользователь вводит цифру «4» и получает информацию о среднем значении минимального и максимального элементов массива. | Программа вычисляет среднее значение минимального и максимального элементов массива, находит все индексы элементов, равные полученному числу: |
| Поиск элементов меньше заданного числа | |
| Пользователь вводит цифру «5», число «а» и получает количество элементов этого числа. | Программа подсчитывает количество элементов меньше заданного пользователем: |
| Поиск элементов больше заданного числа | |
| Пользователь вводит цифру «6», число «b» и получает количество элементов этого числа. | Программа подсчитывает количество элементов больше заданного пользователем: |
| Поиск элемента в отсортированном массиве | |
| Пользователь вводит цифру «7» и значение элемента. | Вычисляется индекс введённого для поиска элемента при помощи алгоритма бинарного поиска, так как массив отсортирован: |
| Обмен элементов массива местами | |
| Пользователь вводит цифру «8» и индексы двух элементов. | Производится обмен элементов по указанным пользователем индексам: |
| ИДЗ: модификация элементов массива и подсчёт кратных разным цифрам | |
| Пользователь вводит цифру «9» и число для модификации массива. | Программа модифицирует нечётные элементы массива на основе ввода пользователя (вычитание значения из элемента и домножение на случайное число от 1 до 10). Подсчитывает количество элементов, кратных от 1 до 9: |

**Выводы.**

В ходе лабораторной работы были изучены алгоритмы сортировки (пузырьком, шейкерная, расческой, вставками и быстрая сортировка) и поиска (линейный и бинарный), а также методы измерения времени выполнения операций с использованием библиотеки <chrono>. Реализованные функции позволили работать с массивами, включая их копирование, вывод и выполнение различных операций, таких как поиск минимальных и максимальных значений, что углубило понимание принципов работы с данными и оптимизации алгоритмов.

Приложение А

рабочий код

#include <iostream>  
#include <chrono>  
#include <ctime>  
  
using namespace std;  
using namespace chrono;  
  
const int N = 100;  
int unsorted\_array[N], sorted\_array[N];  
time\_point<steady\_clock, duration<\_\_int64, ratio<1, 1000000000>>> start\_time, end\_time;  
nanoseconds result\_time;  
  
void bubble\_sort(int \*arr, int n);  
  
void shaker\_sort(int \*arr, int n);  
  
void comb\_sort(int \*arr, int n);  
  
void insertion\_sort(int \*arr, int n);  
  
void quicksort(int \*arr, int end, int begin);  
  
int binary\_search(int \*arr, int value, int start, int end);  
  
void print\_array(int \*arr, int n);  
  
void copy\_array(const int \*src, int \*dest, int n);  
  
void task\_1();  
  
void task\_2();  
  
void task\_3();  
  
void task\_4();  
  
void task\_5();  
  
void task\_6();  
  
void task\_7();  
  
void task\_8();  
  
void task\_9();  
  
  
int main() {  
 srand(time(nullptr));  
 task\_1();  
  
 int choice;  
  
 do {  
 cout << "Choose a task to execute (1-9) or 0 to exit: ";  
 cin >> choice;  
 system("cls");  
  
 switch (choice) {  
 case 1:  
 task\_1();  
 break;  
 case 2:  
 task\_2();  
 break;  
 case 3:  
 task\_3();  
 break;  
 case 4:  
 task\_4();  
 break;  
 case 5:  
 task\_5();  
 break;  
 case 6:  
 task\_6();  
 break;  
 case 7:  
 task\_7();  
 break;  
 case 8:  
 task\_8();  
 break;  
 case 9:  
 task\_9();  
 break;  
 case 0:  
 cout << "Exiting the program." << endl;  
 break;  
 default:  
 cout << "Invalid choice. Please select a number from 0 to 8." << endl;  
 break;  
 }  
  
 } while (choice != 0);  
  
}  
  
void bubble\_sort(int \*arr, int n) {  
 bool swapped;  
  
 for (int i = 0; i < n - 1; i++) {  
 swapped = false;  
  
 for (int j = 0; j < n - 1 - i; j++) {  
 if (arr[j] > arr[j + 1]) {  
 swap(arr[j], arr[j + 1]);  
 swapped = true;  
 }  
 }  
  
 if (!swapped) {  
 break;  
 }  
 }  
}  
  
void shaker\_sort(int \*arr, int n) {  
 bool swapped = true;  
 int start = 0;  
 int end = n - 1;  
  
 while (swapped) {  
 swapped = false;  
  
 for (int i = start; i < end; ++i) {  
 if (arr[i] > arr[i + 1]) {  
 swap(arr[i], arr[i + 1]);  
 swapped = true;  
 }  
 }  
  
 if (!swapped) {  
 break;  
 }  
  
 swapped = false;  
 --end;  
  
 for (int i = end - 1; i >= start; --i) {  
 if (arr[i] > arr[i + 1]) {  
 swap(arr[i], arr[i + 1]);  
 swapped = true;  
 }  
 }  
  
 ++start;  
 }  
}  
  
void comb\_sort(int \*arr, int n) {  
 const double shrinkFactor = 1.247;  
 int gap = n;  
 bool swapped = true;  
  
 while (gap > 1 || swapped) {  
 gap = gap / shrinkFactor;  
 if (gap < 1) {  
 gap = 1;  
 }  
  
 swapped = false;  
  
 for (int i = 0; i + gap < n; i++) {  
 if (arr[i] > arr[i + gap]) {  
 swap(arr[i], arr[i + gap]);  
 swapped = true;  
 }  
 }  
 }  
}  
  
void insertion\_sort(int \*arr, int n) {  
 int i, key, j;  
 for (i = 1; i < n; i++) {  
 key = arr[i];  
 j = i - 1;  
  
 while (j >= 0 && arr[j] > key) {  
 arr[j + 1] = arr[j];  
 j = j - 1;  
 }  
 arr[j + 1] = key;  
 }  
}  
  
void quicksort(int \*arr, int end, int begin) {  
 int mid;  
 int f = begin;  
 int l = end;  
 mid = arr[(f + l) / 2];  
 while (f < l) {  
 while (arr[f] < mid) f++;  
 while (arr[l] > mid) l--;  
 if (f <= l) {  
 swap(arr[f], arr[l]);  
 f++;  
 l--;  
 }  
 }  
 if (begin < l) quicksort(arr, l, begin);  
 if (f < end) quicksort(arr, end, f);  
}  
  
int binary\_search(int \*arr, int value, int start, int end) {  
 if (end >= start) {  
 int mid = start + (end - start) / 2;  
  
 if (arr[mid] == value) {  
 return mid;  
 }  
  
 if (arr[mid] > value) {  
 return binary\_search(arr, value, start, mid - 1);  
 }  
  
 return binary\_search(arr, value, mid + 1, end);  
 }  
  
 return -1;  
}  
  
void print\_array(int \*arr, int n) {  
 for (int i = 0; i < n; ++i) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
}  
  
void copy\_array(const int \*src, int \*dest, int n) {  
 for (int i = 0; i < n; ++i) {  
 dest[i] = src[i];  
 }  
}  
  
void task\_1() {  
 cout << "--- task #1 ----" << endl;  
 for (int i = 0; i < N; ++i) {  
 unsorted\_array[i] = rand() % 199 - 99;  
 }  
  
 cout << "Unsorted array:" << endl;  
 print\_array(unsorted\_array, N);  
  
 cout << "Sorted array:" << endl;  
 copy\_array(unsorted\_array, sorted\_array, N);  
 quicksort(sorted\_array, N - 1, 0);  
 print\_array(sorted\_array, N);  
}  
  
void task\_2() {  
 cout << "--- task #2 ----" << endl;  
 copy\_array(unsorted\_array, sorted\_array, N);  
 start\_time = steady\_clock::*now*();  
 bubble\_sort(sorted\_array, N);  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Bubble sort, ns:\t" << result\_time.count() << endl;  
  
 copy\_array(unsorted\_array, sorted\_array, N);  
 start\_time = steady\_clock::*now*();  
 shaker\_sort(sorted\_array, N);  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Shaker sort, ns:\t" << result\_time.count() << endl;  
  
 copy\_array(unsorted\_array, sorted\_array, N);  
 start\_time = steady\_clock::*now*();  
 comb\_sort(sorted\_array, N);  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Comb sort, ns :\t" << result\_time.count() << endl;  
  
 copy\_array(unsorted\_array, sorted\_array, N);  
 start\_time = steady\_clock::*now*();  
 insertion\_sort(sorted\_array, N);  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Insertion sort, ns:\t" << result\_time.count() << endl;  
  
 copy\_array(unsorted\_array, sorted\_array, N);  
 start\_time = steady\_clock::*now*();  
 quicksort(sorted\_array, N - 1, 0);  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Quick sort, ns: \t" << result\_time.count() << endl;  
}  
  
void task\_3() {  
 cout << "--- task #3 ----" << endl;  
 copy\_array(unsorted\_array, sorted\_array, N);  
 quicksort(sorted\_array, N - 1, 0);  
  
 int min = 100, max = -100;  
  
 start\_time = steady\_clock::*now*();  
 for (int i = 0; i < N; ++i) {  
 if (min > unsorted\_array[i]) {  
 min = unsorted\_array[i];  
 }  
 if (max < unsorted\_array[i]) {  
 max = unsorted\_array[i];  
 }  
 }  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Min: " << min << ", max: " << max << endl;  
 cout << "Unsorted search min and max time, ns: " << result\_time.count() << endl;  
  
 start\_time = steady\_clock::*now*();  
 min = sorted\_array[0];  
 max = sorted\_array[N - 1];  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Min: " << min << ", max: " << max << endl;  
 cout << "Sorted search min and max time, ns: " << result\_time.count() << endl;  
}  
  
void task\_4() {  
 cout << "--- task #4 ----" << endl;  
 copy\_array(unsorted\_array, sorted\_array, N);  
 quicksort(sorted\_array, N - 1, 0);  
 int min = sorted\_array[0], max = sorted\_array[N - 1], count = 0;  
 int average = (max + min) / 2;  
 cout << "Average min and max value: " << average << endl;  
 cout << "Indexes with same value, unsorted array: ";  
 start\_time = steady\_clock::*now*();  
 for (int i = 0; i < N; ++i) {  
 if (unsorted\_array[i] == average) {  
 cout << i << " ";  
 count++;  
 }  
 }  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << endl << "Indexes count: " << count << endl;  
 cout << "Time spent in unsorted array, ns: " << result\_time.count() << endl;  
  
 count = 0;  
 cout << "Indexes with same value, sorted array: ";  
 start\_time = steady\_clock::*now*();  
 for (int i = 0; sorted\_array[i] <= average; ++i) {  
 if (sorted\_array[i] == average) {  
 cout << i << " ";  
 count++;  
 }  
 }  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << endl << "Indexes count: " << count << endl;  
 cout << "Time spent in sorted array, ns: " << result\_time.count() << endl;  
}  
  
void task\_5() {  
 cout << "--- task #5 ----" << endl;  
 copy\_array(unsorted\_array, sorted\_array, N);  
 quicksort(sorted\_array, N - 1, 0);  
 int a, counter = 0;  
 print\_array(sorted\_array, N);  
 cout << "Enter number to count less elements: ";  
 cin >> a;  
 while (sorted\_array[counter++] < a);  
 cout << "Result: " << --counter << endl;  
}  
  
void task\_6() {  
 cout << "--- task #6 ----" << endl;  
 copy\_array(unsorted\_array, sorted\_array, N);  
 quicksort(sorted\_array, N - 1, 0);  
 int b, counter = N - 1;  
 print\_array(sorted\_array, N);  
 cout << "Enter number to count greater elements: ";  
 cin >> b;  
 while (sorted\_array[counter--] > b);  
 cout << "Result: " << N - counter - 2 << endl;  
}  
  
void task\_7() {  
 cout << "--- task #7 ----" << endl;  
 copy\_array(unsorted\_array, sorted\_array, N);  
 quicksort(sorted\_array, N - 1, 0);  
 int number, index = -1;  
 print\_array(sorted\_array, N);  
 cout << "Enter number to search in array: ";  
 cin >> number;  
  
 start\_time = steady\_clock::*now*();  
 for (int i = 0; i < N; ++i) {  
 if (sorted\_array[i] == number) {  
 index = i + 1;  
 break;  
 }  
 }  
 end\_time = steady\_clock::*now*();  
  
 if (index == -1) {  
 cout << "Not found" << endl;  
 } else {  
 cout << "Found at " << index << " position" << endl;  
 }  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Time spent, simple search, ns: " << result\_time.count() << endl;  
  
 start\_time = steady\_clock::*now*();  
 index = binary\_search(sorted\_array, number, 0, N - 1);  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Time spent, binary search, ns: " << result\_time.count() << endl;  
}  
  
void task\_8() {  
 cout << "--- task #8 ----" << endl;  
 cout << "Array:" << endl;  
 print\_array(unsorted\_array, N);  
 cout << "Enter two indexes for swap: ";  
 int i1, i2;  
 cin >> i1 >> i2;  
 start\_time = steady\_clock::*now*();  
 swap(unsorted\_array[i1], unsorted\_array[i2]);  
 end\_time = steady\_clock::*now*();  
 result\_time = duration\_cast<nanoseconds>(end\_time - start\_time);  
 cout << "Time spent for swap, ns: " << result\_time.count() << endl;  
 cout << "New array:" << endl;  
 print\_array(unsorted\_array, N);  
}  
  
void task\_9() {  
 cout << "--- individual homework #9 ----" << endl;  
 cout << "Array:" << endl;  
 print\_array(unsorted\_array, N);  
 cout << "Enter number: ";  
 int number;  
 cin >> number;  
 for (int i = 1; i < N; i += 2) {  
 unsorted\_array[i] -= number;  
 unsorted\_array[i] \*= rand() % 9 + 1;  
 }  
 cout << "New array:" << endl;  
 print\_array(unsorted\_array, N);  
 cout << "Number of elements that are divisible by:" << endl;  
 for (int i = 1; i < 10; ++i) {  
 int counter = 0;  
 for (int j = 0; j < N; ++j) {  
 if (unsorted\_array[j] % i == 0) {  
 counter++;  
 }  
 }  
 cout << i << " - " << counter << endl;  
 }  
}